

AD-A259 145



AFIT/GCS/ENG/92D-10

AN ENHANCED USER INTERFACE
FOR THE SABER WARGAME

THESIS

Donald Ray Moore
Captain, USAF

AFIT/GCS/ENG/92D-10

DTIC
ELECTE
JAN 1 1993
S E D

93-00187



Approved for public release; distribution unlimited

93 1 04 102

AN ENHANCED USER INTERFACE
FOR THE SABER WARGAME

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

Donald Ray Moore, B.S.
Captain, USAF

December, 1992

DTIC QUALITY INSPECTED 5

Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

Approved for public release; distribution unlimited

Acknowledgements

I would like to thank several people whose assistance and support have made this thesis possible. First, I wish to thank my thesis advisor, Maj Mark Roth, whose patient guidance and insightful suggestions helped keep me on track. Next, I wish to thank my readers, MAJ Eric Christensen and Mr. Michael Garrambone, for their technical assistance on the the thesis project itself and their feedback on my thesis draft. I also wish to thank the following fellow students: 1Lt Scott Douglass, who helped keep me focused on the “big picture” instead of distracted by the details, and Capts Mike Lightner and Jeff Heath, who patiently listened to the accounts of my trials and tribulations throughout the project.

I also wish to thank my family for patiently enduring the last eighteen months, and I apologize for having put them through it. I thank my daughter, Shauna, for giving me a hug when I needed a hug, and I thank my son, Andrew, for making me play when I needed to play. Finally, I owe a special “thank you” to my darling wife, Debra, for comforting me and for seldom complaining about the loss of my companionship, even though she often had good reason to do so.

Donald Ray Moore

Table of Contents

	Page
Acknowledgements	ii
Table of Contents	iii
List of Figures	vii
List of Tables	viii
Abstract	ix
I. Introduction	1
1.1 Overview	1
1.2 Background	2
1.2.1 Purpose.	2
1.2.2 Development History.	2
1.3 Problem	5
1.4 Research Objectives	6
1.5 Assumptions	6
1.6 Standards	7
1.7 Approach/Methodology	8
1.8 Materials and Equipment	9
1.9 Thesis Overview	9
II. Literature Review	10
2.1 Introduction	10
2.2 User Interface Design	10
2.2.1 Evolution of the User Interface.	10

	Page
2.2.2 Goals of User Interface Design.	11
2.2.3 Elements of User Interface Design.	12
2.3 The X Window System	15
2.3.1 Architecture of the X Window System.	16
2.3.2 Libraries of the X Window System.	18
2.4 Ada to X Windows Interfaces.	19
2.4.1 Ada Bindings to X Windows.	20
2.4.2 Ada Implementations of X Windows.	22
2.5 Summary	23
III. Approach	24
3.1 Introduction	24
3.2 Development and Integration of the Processors	24
3.2.1 Combined Methodology.	24
3.3 Replacement of the Ada to X Windows Interface	26
3.4 Summary	27
IV. Requirements Specification	29
4.1 Pre-Processor	29
4.1.1 Standard Mission Requirements.	30
4.1.2 Aircraft Beddown Mission Requirements.	30
4.1.3 Supply Mission Requirements.	32
4.1.4 Aircraft Mission Requirements.	34
4.1.5 Land Unit Mission Requirements.	36
4.2 Summary	37
V. Integrated Design of the Processors	38
5.1 Introduction	38
5.2 Engineering the Design	39

	Page
5.2.1 Reverse Engineering the Existing Designs.	40
5.2.2 Forward Engineering the Integrated Design.	42
5.3 Summary	44
VI. Implementation and Investigation	45
6.1 Introduction	45
6.2 Integrated Implementation of the Processors	45
6.3 Integrated Implementation of the Pre-Processor	46
6.3.1 Help.	47
6.3.2 List Widget.	48
6.3.3 Manager Widget.	49
6.3.4 Mission Board.	49
6.4 Investigation of Replacing the Ada to X Windows Interface . . .	55
6.4.1 Comparison of the STARS and SERC Identifiers.	57
6.4.2 Incremental Replacement of the STARS Identifiers.	61
6.5 Summary	63
VII. Conclusions and Recommendations	64
7.1 Conclusions	64
7.2 Recommendations	65
7.3 Summary	66
Appendix A. Saber Object Class Descriptions	67
A.1 Application Classes	67
A.1.1 Airbase Class.	67
A.1.2 Aircraft Mission Class.	69
A.1.3 Animation Controller Class.	71
A.1.4 Day End Controller Class.	74
A.1.5 Forces Class.	75

	Page
A.1.6 Help Class.	75
A.1.7 Hexboard Class.	78
A.1.8 Game Player Class.	80
A.1.9 Ground Unit Class.	81
A.1.10 Main Controller Class.	84
A.1.11 Main Procedure Class.	85
A.1.12 Mission Board Class.	85
A.1.13 Report Class.	86
A.1.14 Terrain Class.	87
A.2 Motif Classes	89
A.2.1 List Widget Class.	89
A.2.2 Manager Widget Class.	90
A.2.3 Menubar Class.	93
A.2.4 Toggle Button Board Class.	94
Appendix B. STARS Identifiers Imported by the Saber User Interface	96
B.1 Boeing Binding Library Packages	96
B.2 SAIC Binding Library Packages	99
Bibliography	100
Vita	102

List of Figures

Figure	Page
1. Mann's Typical Combat Model	4
2. The Computer and Language Elements	13
3. The X Client-Server Model	17
4. Typical X Windows Configuration	20
5. Klabunde's Saber User Interface Design Methodology	25
6. Integrated User Interface Object Diagram	41
7. Integrated User Interface Ada Module Diagram	46
8. Beddown Mission Board Help	48
9. Aircraft Type Help	49
10. Source Airbase Error Dialog	50
11. Select a Mission Information Dialog	50
12. Quit Mission Board Warning Dialog	50
13. Beddown Mission Board	52
14. Airbase Help	53
15. Aircraft Quantity Help	54
16. Delete Mission Warning Dialog	54
17. Save Missions Warning Dialog	56
18. Execute Missions Warning Dialog	56
19. Quit with Unsaved Missions Warning Dialog	56
20. User Interface Relationship to the Ada Bindings	57
21. Example Entry from the STARS to SERC Conversion Guide	60
22. XT.Arg.List Entry from the STARS to SERC Conversion Guide	62

List of Tables

Table	Page
1. Horton's Saber Mission Matrix	34
2. Conversion Level of Effort	61

Abstract

This thesis is part of an on-going effort by the Air Force Institute of Technology to develop a computer-based, theater-level wargame for the Air Force Wargaming Center at Maxwell AFB, AL. The wargame, Saber, is intended to augment the education the students receive at the Air War College and the Air Command and Staff College in the employment of air and ground power.

This thesis documents the integrated design and implementation of the two components of the Saber user interface: the pre-processor and the post-processor. Although previous thesis students designed and implemented substantial portions of the user interface, a fully operational interface was not completed for the end-user. In particular, the pre-processor design and implementation was incomplete and the user interface was constrained by its Ada to X Windows interface. Furthermore, the design and implementation of the two processors now needed to be integrated.

The user interface was designed using object-oriented programming techniques. As necessary, reverse engineering techniques were used to extract the design of the existing implementation. Although the user interface application is implemented in the Ada programming language, it relies upon several software libraries including the X Window System and the OSF/Motif widget. Furthermore, software libraries from the Software Technology for Adaptable Reliable Systems (STARS) Foundation were used to provide the interface (the binding) between the Ada application software and the X Window System including Motif. This thesis also investigates the feasibility of replacing the STARS' bindings with those developed by the Systems Engineering Research Corporation.

AN ENHANCED USER INTERFACE FOR THE SABER WARGAME

I. Introduction

1.1 Overview

This thesis is part of an on-going effort by the Air Force Institute of Technology to develop a computer-based, theater-level wargame for the Air Force Wargaming Center at Maxwell AFB, AL. The wargame, Saber, is intended to augment the education the students receive at the Air War College and the Air Command and Staff College in the employment of air and ground power. This thesis documents the integrated design and implementation of the two components of the Saber user interface: the pre-processor and the post-processor. The pre-processor provides the capability to submit instructions for the game's assets; the post-processor provides the capability to display the results of those instructions through graphical animation and textual reports. Although previous thesis students designed and implemented substantial portions of the user interface, a fully operational interface was not completed for the end-user. In particular, the pre-processor design and implementation was incomplete and the user interface was constrained by its Ada to X Windows interface. Furthermore, the design and implementation of the two processors now needed to be integrated.

The user interface was designed using object-oriented programming techniques. As necessary, reverse engineering techniques were used to extract the design of the existing implementation. Although the user interface application is implemented in the Ada programming language, it relies upon several software libraries including the X Window System and the OSF/Motif widget. Furthermore, software libraries from the Software Technology for Adaptable Reliable Systems (STARS) Foundation were used to provide the interface (the binding) between the Ada application software and the X Window Sys-

tem including Motif. This thesis also investigated the feasibility of replacing the STARS' bindings with those developed by the Systems Engineering Research Corporation.

1.2 Background

Saber is a computer-based, theater-level wargame that provides an environment for modeling the air and land conflict between two military forces. Given a database scenario that defines the initial status of two military forces within a geographical theater, two teams of players can submit instructions through the user interface to their respective forces. The model then executes those instructions, simulates the outcome based on the capabilities of each force, and provides feedback to the players on the results of their decisions. During the execution phase, Saber "models conventional, nuclear, and chemical warfare at the aggregated air and ground forces level with the effects of logistics, satellites, weather, and intelligence represented"[12:1].

Since the individual elements of the air and land forces are aggregated into larger units, players can control all units by submitting instructions directly to the headquarters of the large (aggregate) units. Saber can model the conflict between two military forces regardless of scenario location. Also, Saber is interactive except that players cannot modify their instructions during the model execution phase.

1.2.1 Purpose. Saber is being developed by the Air Force Institute of Technology (AFIT) for the Air Force Wargaming Center (AFWC) at Maxwell AFB, AL. The players of the wargame will be the faculty and students at the Air War College and the Air Command and Staff College at Maxwell AFB. Saber will augment the education the students receive in the employment—planning, preparation, and execution—of air and ground power. Specifically, it will provide the force information necessary for the players to make decisions about the employment of the forces and a vehicle for implementing those decisions[12, 15].

1.2.2 Development History. Saber will integrate two sub-models—a land combat model and an air combat model. The land combat model was developed by Capt Marlin

Ness in 1990 to replace the one used by AFWC during their Theater War Exercises. "The model simulates doctrinal planning and decision making operations conducted at the Army Group level" [17:x]. "The lowest unit of aggregation for the new land model are divisions and major corps non-divisional units (brigades and regiments)"[17:15]. Units are located upon and move across a "geographical" grid of hexagons. The rate of movement of a unit within a given hex is affected by the terrain, obstacles, and weather associated with that hex. Attrition occurs whenever opposing forces are located in adjacent hexes and is based primarily on the *firepower score*, a measure of combat capability, of the units involved. The activities of the units are modeled as discrete events and occur in fixed time steps. In other words, the status of the forces is updated at fixed time intervals[17]. "The output of Ness' model consisted of rather lengthy and somewhat cryptic reports"[12:2].

Ness used object-oriented design techniques and implemented the land model using the Ada programming language. He used the Oracle database management system to generate the scenario database and some reports. The Oracle database must be downloaded to flat files for execution by the model [17].

In 1991, Captain William Mann laid the foundation for the development of a new air model to replace the one used by AFWC during their Theater Warfare Exercises. Mann's first objective was to "determine how the US Air Force doctrinally conducts a tactical air theater campaign and to link it to the model"[15:6]. His second objective was to "assemble the model and its components"[15:6] similar to the plan depicted in Figure 1. Saber would evolve from the integration of the new air model and Ness' land model. Mann defined the entities required for the air model, including bases and aircraft, and expanded the definition of the units in the land model to support the integration of the two models. Furthermore, he developed "the algorithms and the related formulas for programmers to construct an object oriented computer simulation"[15:viii].

Later in 1991, three other efforts progressed toward development of the air model and the eventual integration of the air and land models. Captain Gary Klabunde's task was to create a graphical post-processor for the wargame. He produced a graphical user interface capable of presenting a geographical display of the air and land battle in either of two ways—a static display of the current situation or an animated display of the previous

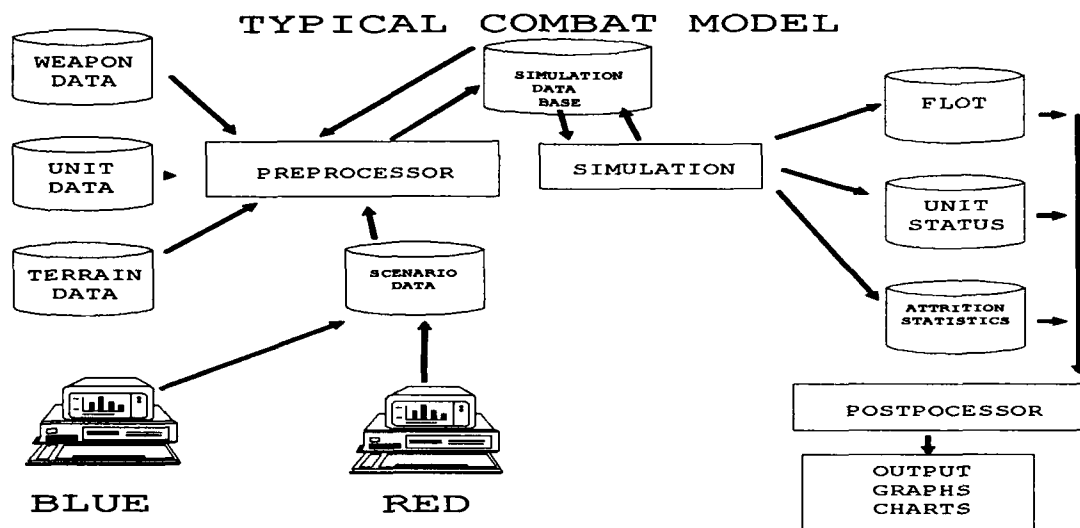


Figure 1. Mann's Typical Combat Model[15:7]

day's activities. Besides displaying the location and status of forces, the post-processor can also be used to portray weather and to generate hard-copy or on-screen status reports[12].

Using object-oriented design techniques, Klabunde implemented the graphical post-processor using the Ada programming language, the X Window System, and the Open Software Foundation(OSF)/Motif widget set. This approach led to a product that is not only portable across a number of hardware platforms but also extensible to ease modification. To simplify the interface between the application program (written in Ada) and the windowing system (written in C), Klabunde incorporated an Ada to X Windows interface consisting of Ada bindings to X Windows from several sources, including Science Applications International Corporation, Boeing Corporation, and the AFWC.

Captain Andre Horton's goal was to "document the design and implementation of a graphical user interface and relational database management system" [10:3] for Saber. Horton designed a pre-processor to accept the four areas of player input described by Mann—aircraft beddown, transportation of supplies, instructions to land units, and aircraft and missile missions. The heart of his task however lay in the development of an integrated database management system for the wargame. He accomplished that task and constructed a realistic scenario based on the Korean theater[10].

Like his predecessors, Horton used object-oriented techniques to design the user interface and the database management system. He incorporated his implementation of a user interface with Klabunde's user interface and implemented a database using the Oracle relational database management system. Horton also provided utilities to transfer the database to and from the relational database system and the flat files required by the wargame for execution[10].

Captain Christine Sherry's objective was to "combine Ness' prototype land battle model with Mann's conceptual air battle description to build an integrated theater level computer wargame"[22:2]. She identified and resolved several hindrances to the integration of the land and air models including reorientation of the hex grid, changing the way the actual weather is determined, and generating a history file of each day's activities for the post-processor. Although Sherry did not achieve a functional product, she did design and implement the majority of Mann's algorithms and she redesigned Ness' land model where necessary for integration[22].

Sherry used object-oriented techniques to design an integrated air and land model. She implemented the model using the Ada programming language [22].

1.3 Problem

Although the efforts of previous thesis students have contributed immensely to achieving the overall objective of an integrated land and air combat model, the model was not ready for operation by the end-user, the AFWC. In particular, the pre-processor was incomplete and the graphical user interface was constrained by the current Ada to X Windows

interface which, among other things, does not provide a complete interface to X Windows and is sparsely documented. Also, implementation of the pre-processor required refinement of the database design to complete the integration effort.

1.4 Research Objectives

The following specific objectives supported the overall goal of developing an integrated graphical user interface and database:

1. Complete the pre-processor design and implementation. Commensurate with this, the design and implementation of the two processors will be integrated. These two steps will provide the means for the user to submit instructions to the game's assets and will eliminate any inconsistency or redundancy between the two processors.
2. Investigate the feasibility of replacing the current Ada to X Windows interface, which uses Ada bindings to X Windows developed by Science Applications International Corporation and Boeing Corporation, with bindings from Systems Engineering Research Corporation. This has the potential of enhancing the maintainability and functionality of the model.

1.5 Assumptions

The research, design, and implementation efforts of this project are based on the following assumptions:

1. "Command, control, and communications would be modeled by the player interaction in the game and not by the computer simulation"[17:5].
2. "Verification and validation of the new land (and air) battle would be conducted by the Air Force Wargaming Center"[15:6].
3. "Naval operations will not be attempted to be modeled at this time"[15:6].
4. "The wargame will use only unclassified data and algorithms" [15:6].
5. "The new game will have the flexibility to play any scenario or theater of operations"[15:6].

6. "The code developed by Ness, Sherry, and Horton correctly creates and updates the databases"[12:5].
7. The graphical user interface and database will continue to be developed and executed on the Sun Sparc Station II or compatible workstation.
8. The graphical user interface and database will continue to be developed using the Ada programming language where feasible, the C programming language, the X Window System, the OSF/Motif widget set, and the Oracle relational database management system.
9. All player input/output and simulation execution will be controlled through the graphical user interface.

1.6 Standards

This research, design, and implementation efforts of this project adhere to the following standards:

1. Where applicable, documentation "shall be in accordance with the guidelines developed for AFIT by Dr. Thomas Hartrum" [12:6]. A consistently documented system makes future development and maintenance of the system easier.
2. The style of the Ada source code will be based on the programming practices described in the Software Productivity Consortium's *Ada Quality and Style* guide[23]. This makes the source code more readable and understandable resulting in a more maintainable system.
3. "The Ada *use* clause will only be used if absolutely necessary"[12:6]. Without the *use* clause, the developer must explicitly name the source of any variables declared outside of the current file. This reduces the possibility of ambiguous variable names and the errors associated with them. This practice also provides a trail to the source of the variable declarations which makes future development and maintenance of the system easier.

4. "There will be a consistent use of naming and capitalization of variables and reserved words"[12:6]. This makes them stand-out and makes future development and maintenance of the system easier.
5. "The behavior of the user interface will closely follow the *OSF/Motif Style Guide*"[12:6]. This helps ensure the user interface is "user-friendly". Since Motif is a common user interface tool, this helps ensure the user is presented with a familiar user interface style and results in a reduced learning curve and error rate by the user.

1.7 Approach/Methodology

The general sequence of events required to achieve the stated objectives are outlined below:

1. Analyze the problem domain/literature review. This step includes a survey of wargaming in general and Saber's development history. However, since many of the implementation decisions have already been made, the focus of the review is on user interfaces, the Ada programming language, the C programming language, the X Window System, and Ada to X Windows interfaces. Knowledge of problem area is essential to continued development.
2. Test the functional operation of the graphical user interface and database. Using the existing documentation as a guide, this step helps to identify any inconsistencies between the documentation and the system as well as undocumented deficiencies in the system.
3. Investigate the feasibility of replacing the current Ada to X Windows interface, which uses Ada bindings to X Windows.
4. Identify and consolidate the requirements. This step is based on an in-depth review of the previous thesis projects and consultation with Saber development team. As with any project, a clear understanding of the goal is imperative to attaining that goal.
5. Design and integrate the requirements within the existing high-level object-oriented design. Since a high-level design already exists, changes to it are kept to a minimum.

6. Implement the system using iterative “risk assessment, detailed design, coding, and testing” to provide “a way to control and measure changes being made to the system as it is being developed” [12:6].

1.8 Materials and Equipment

Saber requires the following development environment: a Sun Sparc Station II or compatible workstation, the Verdex Ada programming language, the C programming language, the OSF/Motif widget set, the X Window System, and SERC's SA-Motif.

1.9 Thesis Overview

Chapter II is a literature review of user interface design, the X Window System, and Ada to X Windows interfaces. Chapter III describes the approach used to develop and integrate the pre-processor with the post-processor as well as the approach used to investigate the feasibility of replacing the current Ada to X Windows interface. Chapter IV clearly identifies and consolidates the requirements driving the objective of integrating the pre-processor and the post-processor. Chapter V describes the integrated design of the pre-processor and the post-processor. Chapter VI discusses the integrated implementation of the pre-processor and post-processor as well as an investigation of the feasibility of replacing the current Ada to X Windows interface. Finally, Chapter VII summarizes the thesis effort and presents conclusions and recommendations.

II. Literature Review

2.1 Introduction

The purpose of this chapter is to condense the knowledge essential for continued development of Saber. Since they are integral components of this project, this chapter describes the user interface design, the X Window System, and the Ada to X Windows interfaces.

2.2 User Interface Design

This section identifies the goals of user interface design and the factors that should be considered by the user interface designer to ensure an effective, efficient interface. When humans fail to communicate effectively and efficiently with each other, they grow frustrated and may accomplish little. In fact, disastrous results may occur. Similar results can occur when the communication between humans and computers is deficient in some way. The environment in which human-computer communication occurs is commonly referred to as the user interface. Klabunde defines the *user interface* as "the component of the application through which the user's actions are translated into one or more requests for services of the applications, and that provides feedback concerning the outcome of the requested actions"[12:15]. Consequently, user interface design should be a prime concern for the designer of any computer system.

In the following discussion of user interface design, a brief history is presented of how the requirements of the user evolved and why a well-designed user interface became so important. Next, the goals of user interface design are identified. Finally, the three main elements of the user interface—the human, the computer, and the language—are examined.

2.2.1 Evolution of the User Interface. User interface design has only recently become an important issue in computer science. Kross writes that in the early years of computing, the only users of computers were the computer scientists themselves[13]. Communication was achieved directly through a control panel of the computer. Although the

means of user interface was quite limited, it was effective and well understood by the user. As the capabilities of computer hardware and software advanced and the cost of hardware decreased, the number of computer users also increased. The new generation of users often had different skills and objectives than their predecessors.

Consequently, the development of “user-friendly” interfaces emerged as a consideration to the designer of any computer system. Designers soon recognized that “the quality of the user interface often determines whether users enjoy or despise a system, whether the designers of the system are praised or damned, whether a system succeeds or fails. . .”[8:347]. In fact, regardless of the functional capability of the system, it became apparent that users will often judge a system solely on the quality of the user interface[24].

2.2.2 Goals of User Interface Design. In his definition of *ergonomics*, Bullinger describes general goals that are applicable to user interface design:

Ergonomics uses scientific methods to describe (a) human attributes and (b) work. Work is described in terms of its processes and environment, where it is done and how it is organised. The object is to adjust the nature of work to suit human characteristics. Our principal aim is, as far as possible, to avoid subjecting human beings to stress in their work and to ensure that they derive the maximum benefit from their abilities, skills, tools and resources. [4:13]

Specifically, Foley identifies five key goals of user interface design[8]:

- *Increase the speed of learning by the user.* This is especially important for systems used infrequently by any one person.
- *Increase speed of use.* This is especially important for systems used for extended periods of time by experienced persons.
- *Reduce the error rate.* This increases speed of learning and speed of use.
- *Encourage rapid recall of how to use the interface.* This also increases the speed of learning and speed of use.
- *Increase the attractiveness of the interface.* This results in a more satisfied user. However, repeated studies have shown that the productivity of a user is not directly related to the personal preferences of the user.

2.2.3 Elements of User Interface Design. To achieve the goals of user interface design, the user interface designer must consider the issues associated with each of the elements of the user interface—the human, the computer, and the language. Human issues include the physical and psychological aspects of the user. Computer issues revolve around the technological capabilities and the types of devices that exist. Language issues involve characteristics and formats of the language that are essential.

2.2.3.1 The Human Element. Although user interface design is not a precise science, “there are some specific do’s and don’ts that, if applied creatively, can help focus attention on the *human factors*, also called the *ergonomics*, of an interactive system”[8:391].

Many ergonomic factors—resources and equipment, the place of work, the environment, and the organization of work[4]—may be outside the control of the user interface designer. Even so, the user interface designer must strive to eliminate or reduce as much as possible any sources of physical stress for the user. For example, tasks that call for repeated, continuous physical action are likely to be stressful. Tasks that require the use of a mouse, a joystick, and especially a screen sensitive light pens are tiresome. Finally, although the impact may be minimal, the mouse and joystick also require new forms of hand-eye coordination by the user[8].

Regardless of the user’s background or experience, there are several sources of psychological stress for the user interface designer to consider. “These include information overloading, task complexity, system response time and the degree of control over the system which the user is allowed”[24:207]. To avoid overloading the user’s short term memory, information should be presented in small units rather than groups. Similarly, task complexity should be reduced by breaking large tasks into small tasks that can be completed sequentially. Although standard response times may not be possible, the user should at least have some indication of how long a task will take. Finally, since the desire of the user to exercise control over a task is commensurate with the skill level of the user, the user interface should be flexible enough to accommodate a range of skill levels [24].

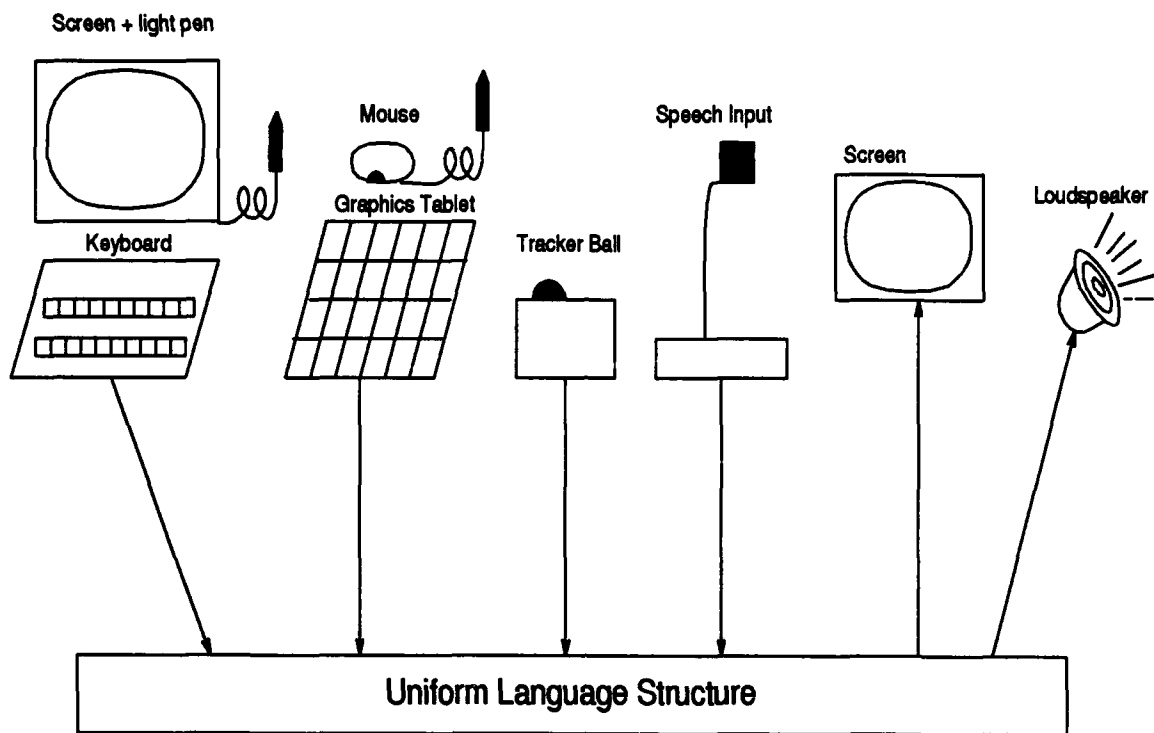


Figure 2. The Computer and Language Elements[4]

2.2.3.2 The Computer Element. “The technical evolution of these devices, such as those in Figure 2 has been the main factor in setting the limitations for human-computer interaction and in deciding what freedom the software ergonomist has in designing suitable interfaces”[4:102]. Common interface devices include locator devices and keyboard devices. More exotic devices, some of which are still experimental, include voice recognizers and voice synthesizers. Locator devices such as data tablets are useful for digitizing drawings while mice are useful for positioning the cursor anywhere on the screen. Keyboard devices such as the standard QWERTY keyboard are useful for entering alphanumeric information to the computer. Voice recognizers respond to the spoken word and are useful for freeing the hands of the user for other tasks. Voice synthesizers generate sound and are useful for augmenting rather than replacing visual feedback to the user. Since the device selection is often dictated by the type of application or by economics, the user interface designer must often “hope the devices are well designed, and attempt to compensate in software for any hardware deficiencies” [8:358].

2.2.3.3 *The Language Element.* Since the language or “dialogue”[4, 24] element of user interface design appears to have received substantially more attention than the other two elements, it is given greater attention in this paper. Suffice it to say that the languages of humans and computers are distinct. Humans understand strings of alphanumeric characters and symbols whereas computers understand the electrical states of “on” and “off”. “In order for the two to communicate, an intermediate language must be developed which is somewhere between the two extremes, and each entity must perform the necessary translation between their natural and this intermediate form”[19:7].

Interface languages can be categorized as computer-initiated, user-initiated, and hybrids of the former two. Computer-initiated languages like menus and fill-in forms require less training but are not very flexible. User-initiated languages like command-line instructions and query languages are efficient and flexible but take longer to learn. Hybrids, which are some combination of the other two languages plus symbolic icons and pointers, overcome many of the shortcomings of the two previous languages[4, 24]. Since no language is clearly better than another, the user interface designer must base his selection on “all the relevant factors, many of which work directly against each other (the nature of the task, the user profile, the limits of available technology, the cost effectiveness of the project as a whole, etc.)”[4:37].

Sommerville identifies three fundamental principles of user interface design--suit the needs and abilities of the user, be consistent, and provide a built-in help facility[24]. These three principles summarize the characteristics that most authors believe a user interface should exhibit. Molich and Nielson identify nine specific principles of user interface design which contribute to achieving the goals expressed by Bullinger and Foley[16:339].

- *Simple and Natural Dialogue.* Present information in a logical order and avoid unnecessary or seldom-used information.
- *Speak the User's Language.* Tailor the language to use words and concepts familiar to the user.
- *Minimize the User's Memory Load.* Provide instructions that are simple and easily accessible by the user.

- *Be Consistent.* Avoid words or actions that have ambiguous or duplicate meanings.
- *Provide Feedback.* Keep the user informed by providing appropriate feedback within a reasonable time.
- *Provide Clearly Marked Exits.* Permit the user to easily escape from an unwanted state.
- *Provide Shortcuts.* Provide the flexibility for experienced users to bypass lengthy instructions and to reduce the steps to perform lengthy operations.
- *Provide Good Error Messages.* Provide messages that blame the system for its inadequacies, identify the error precisely, and suggest what to do next.
- *Error Prevention.* Prevent problems from occurring in the first place.

Although these specific principles were used to categorize the problems encountered by the users of an example text-based user interface, they are equally applicable to graphic-based user interfaces[18:109]. In fact, graphics-based user interfaces provide the opportunity for not only increased productivity, but also increased errors by the user. Thus, it is even more essential for the user interface designer to adhere to these principles[18:117].

The one principle echoed by virtually all authors is consistency [4, 8, 10, 12, 13, 16, 18, 24]. However, Grudin argues that consistency is potentially harmful because “when user interface consistency becomes our primary concern, our attention is directed away from its proper focus: users and their work” [9:1164]. Although Grudin’s emphasis is that the principle of consistency is not absolute, his article serves to remind the user interface designer that *no principle is absolute*. In other words, the principles mentioned here are simply guidelines and the user interface designer must often make tradeoffs between these principles to satisfy the goals of the task at hand. But, at least the designer knows what is being traded off[18:117].

2.3 The X Window System

This section describes the X Window System and the libraries most commonly used by application programmers to access it. “The X Window System is an industry-standard

software system that allows programmers to develop portable graphical user interfaces" [28:1]. Although this is a very generic definition, it does highlight the two key factors that make the X Window System so attractive. First, portability is a valuable, if not essential, trait for any system. As such, X Windows allows application programs to run on any "hardware that supports the X protocol without modifying, recompiling, or relinking the application"[28:1]. Second, the fact that X Windows is supported by so many vendors promotes the portability of systems and helps to reassure the developer of continued support by industry. In fact, a wide variety of hardware and software manufacturers[28:512] have formed an alliance known as the X Consortium. They control the definition of the system and are committed to using it as the basis for their products. A third factor that makes the X Window System attractive is that it "tries to provide the mechanism to create graphical interfaces, but with no single policy"[11:4]. Consequently, the application programmer can layer almost any widget set (defined later in this section) on top of the X Window System.

2.3.1 Architecture of the X Window System. "The architecture of the X Window System is based on a *client-server* model[28:2] like the one depicted by Klabunde in Figure 3. The server is the process that "is responsible for all input and output devices" [28:2]. Conversely, the client is an "application that uses the facilities provided by the X server"[28:2]. Just as multiple clients can be served by a single server as shown in Figure 3, a single client may be served by multiple servers. Communications between clients and servers conform to a communication's network protocol, the X protocol, which is compatible with a variety of network protocols.

Communications between clients and servers takes the form of messages called requests and events. A *request* is a message sent by a client to a server whenever the client needs to either send information to a device or get information about the status of a device. For example, a client may send the server a request to display a window or to ask if a particular window is currently being displayed. On the other hand, an *event* is a message sent by a server to a client. For example, the server may send the client an event informing the client that the user has pressed a mouse button or moved the mouse pointer.

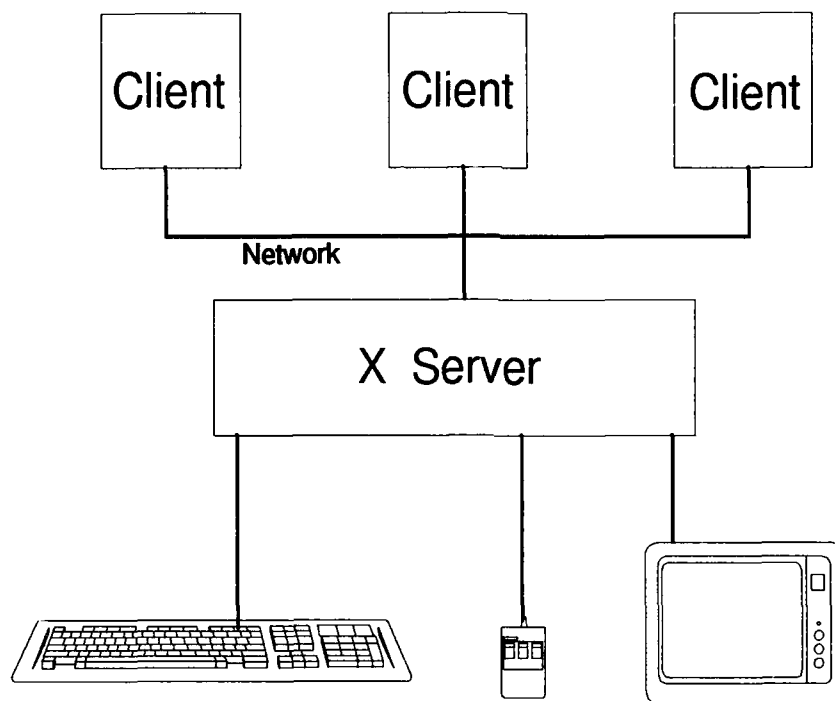


Figure 3. The X Client-Server Model[12:22]

One key point should be made regarding requests and events. A server does not send an event to every client about every action by a user. First of all, a server only sends an event to a given client if the user action that caused the event involves an object created by the client. Secondly, a server usually sends an event to a client only if the client requested to be notified of that type of event when it asked the server to create the object. For example, consider the following simple, if impractical, scenario. Suppose a client application sends a request to a server to create a pushbutton and display it on the screen. Suppose the request also asks the server to send an event to the client whenever the mouse pointer is over the pushbutton and the left mouse button is pressed. Suppose further that the request includes the address of a procedure in the client application called `RemovePushButton` that is to be executed whenever the event occurs. Finally, suppose `RemovePushButton` contains a request from the client to the server to remove the pushbutton from the screen. Thus,

whenever the mouse pointer is over the pushbutton and the left mouse button is pressed, the server in effect initiates the execution of the client procedure `RemovePushButton`. In turn, `RemovePushButton` asks the server to remove the pushbutton from the screen.

One final point should be made about the preceeding example. In X parlance, the procedure `RemovePushButton` is what's commonly referred to as a *callback function* because, in effect, the server "makes a call back to the application-defined function"[28:34]. As seen later, callbacks will play a major role in the implementation of the Saber model.

2.3.2 Libraries of the X Window System. "Although the X server protocol is defined at the level of network packets and byte-streams, programmers generally base applications on libraries that provide an interface to the base window system"[28:11]. The minimal library that an application programmer needs access to is Xlib, which may be augmented with one or more toolkits.

2.3.2.1 Xlib. The X Consortium recognizes Xlib as the standard low-level library interface to the X Window System, and it is the most commonly used. Xlib is written in the C programming language, and it provides effective, complete access to the X Window System. Unfortunately, it can be cumbersome and difficult for the application programmer to use. Consequently, many application programmers prefer to interface with the X Window System through a higher level toolkit designed for use with X.

2.3.2.2 Toolkits. Although a variety of toolkits exist [12:25], Young discusses the X Toolkit which is comprised of a "layer known as the Xt Intrinsics, and a set of user interface components known as widgets" [28:12]. A widget set is a predefined (either by the user or some or some other source) set of objects that can be displayed on a screen, such as pushbuttons and menus. The Xt Intrinsics provides the mechanism for structuring the widgets into a cohesive user interface. Although the X Consortium recognizes the Xt Intrinsics as a standard part of the X Window System, it may recognize other similar interfaces. Finally, the X Consortium has not recognized any widget set as a standard part of the X Window System. Since a standard widget set may restrict the "look and feel" of the application, this is consistent with their goal of providing mechanism, not policy[11:4].

As pointed out by Johnson, OSF/Motif is one of the widget sets commonly chosen by application programmers as a part of their X Toolkit and he lists three advantages to doing so[11:4]:

1. Motif provides a standard interface with a consistent look and feel. Your users will have less work to do in learning other Motif applications, since much of the work learning other Motif applications will translate directly to your applications.
2. Motif provides a very high-level object-oriented library. You can generate extremely complex graphical programs with a very small amount of code.
3. Motif has been adopted by many of the major players in the computer industry. Many of your customers are probably using Motif right now. You'll do a better job selling to them if your applications are also based on Motif.

As seen in Figure 4, an application program may make calls directly to the widget set (in this case Motif), to the Xt Intrinsics, or to Xlib. In this particular example, the Motif widget set may make calls directly to the Xt Intrinsics but not to Xlib. However, the widget set is not prohibited from making calls directly to Xlib.

2.4 Ada to X Windows Interfaces.

This section briefly describes the approaches taken to interface applications programs written in Ada with the X Window System. As noted earlier, applications programs access the X Window System through libraries. Unfortunately, most X Window libraries, including Xlib, the Xt Intrinsics, and OSF/Motif were originally written in the C programming language. Although Ada has the facilities for interfacing with other programming languages, each call to the other language must be specially tailored based on the language being called and the particular call being made. As noted by Klabunde, there have been two approaches to interfacing Ada applications programs with the X Window System: Ada bindings to X Windows and Ada implementations of X Windows[12:27].

Klabunde has already presented a concise, yet thorough, description of both of these approaches. Consequently, the following descriptions will only repeat the information essential for continuity of the discussion and add new information as appropriate. In the

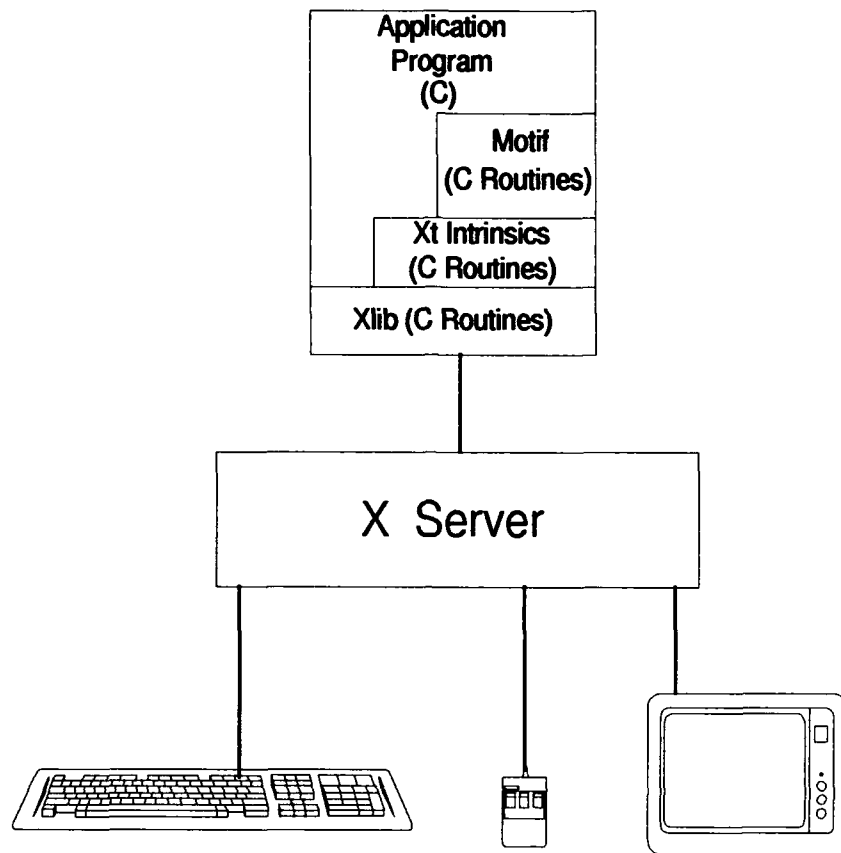


Figure 4. Typical X Windows Configuration[12:26]

following discussion of the Ada to X Windows Interface, each approach to interfacing Ada applications to the C libraries of X is briefly described.

2.4.1 Ada Bindings to X Windows. As noted earlier, Ada has the facilities for interfacing with other programming languages. An Ada binding to X Windows, or more precisely, an Ada binding to a C library of X Windows, uses one of these facilities. In short, an Ada binding to X Windows is an Ada subprogram that is tailored to make a particular call to a subprogram in one of the C libraries of the X Window System. In practice, the Ada application program would call the desired Ada binding. The Ada binding would then perform any transformations that need to occur and call the appropriate subroutine in a C library of the X Window System. Although application programmers could define their own bindings from Ada to the C libraries of the X Window System, the complexity of

the X Window System may make this choice impractical. Fortunately, there exist several sources, public domain and commercial, for Ada to X Windows bindings. We examined the public domain version available from the Software Technology for Adaptable Reliable Systems (STARS) Foundation and a commercial version from the Systems Engineering Research Corporation (SERC). Both bindings were written for the Verdix Ada compiler used for Saber.

2.4.1.1 STARS Ada Bindings to X Windows. During the late 1980's, the STARS Foundation distributed the products of several efforts to develop Ada to X Windows bindings. In one effort, the Science Applications International Corporation (SAIC) developed Ada bindings to the Xlib C routines. In another effort, the Boeing Corporation developed Ada bindings to a large part of the Xt Intrinsics and OSF/Motif and to a small portion of Xlib. Other efforts were also sponsored by STARS. There are advantages and disadvantages to using the STARS bindings. The most obvious advantage is that the application programmer is relieved of the necessity of creating the bindings. Also, these bindings are available in the public domain at no cost. Unfortunately, the bindings do not provide complete access to the X Window System. Consequently, the applications programmer is compelled to either do without some desired functionality or to develop a work-around. Also, as Klabunde discovered, the bindings are sparsely documented and at least one of the bindings is constrained to using a vendor-specific version of Ada, Verdix Ada Development System (VADS) 5.5 or higher[12:27]. He also observed that the bindings appear to have some "bugs", at least in their interaction with Ada tasks. Perhaps the biggest disadvantage to these bindings, however, is the fact that there is no ongoing maintenance of the bindings. Consequently, the applications programmer has no one to turn to for assistance, and the bindings are not updated as new versions of the X Window libraries are released.

2.4.1.2 SERC Ada Bindings to X Windows. More recently, SERC developed a commercial binding, SA-Motif, which provides a complete Ada binding to the the C libraries of the X Window System including Xlib, the Xt Intrinsics, and OSF/Motif. SA-Motif also contains bindings to other commonly used, but non-standard, X Windows li-

braries. SERC's "Xlib and Xt bindings are based on the public domain architecture which was created by STARS for Xlib and Xt [27:1]. Like the STARS bindings, the SERC bindings have their advantages and disadvantages. Besides relieving the applications programmer of the necessity of creating the bindings, the most obvious advantage to the SERC bindings is that they provide complete access to the X Windows libraries. Also, although not guaranteed, ongoing support for the SERC bindings is expected. Further, the SERC bindings are fairly well documented. Finally, the SERC bindings fixed existing bugs in the STARS bindings[27:1]. Unfortunately, the major disadvantage with the SERC bindings is that they are not available free of charge. Furthermore, in certain circumstances, their licensing scheme may prohibit computers that are unlicensed for SA-Motif from executing applications that were developed using SA-Motif. It is understandable that SERC would compel development computers to be licensed for SA-Motif. It is not understandable that they would compel run-time computers to be licensed for SA-Motif. To their credit, they have attempted to define run-time checks in such a way that an application developed using SA-Motif can run on an unlicensed run-time computer, but their scheme may not always produce the desired result[26:1]. Also, only the Ada package specifications for the SERC bindings are available for reference by the applications programmer. Since the Ada package bodies are not available for reference, it is more difficult for the applications programmer to develop application-specific widget sets. Finally, like the STARS bindings, the SERC bindings are constrained to using vendor-specific versions of Ada, either VADS 6.0.3 or Sun Ada 1.0 [25:1].

2.4.2 Ada Implementations of X Windows. Although an Ada implementation of X Windows is not applicable to this thesis effort, the following comments are necessary to make this discussion complete. In at least one case, the STARS Foundation sponsored an effort to develop a partial Ada implementation of X Windows, or more precisely, an Ada implementation of one of the C libraries of X Windows. In 1989, "Unisys Corporation developed an Ada implementation ...of the X11R3 version of the Xt Intrinsics"[12:30] called Ada/Xt. Since Ada/Xt accessed Xlib through an updated version of SAIC's Ada bindings to X Windows, it would appear to have inherited the advantages and disadvantages of SAIC's Ada bindings.

2.5 *Summary*

This chapter discussed three of the integral components of this project: user interface design, the X Window System, and Ada to X Windows interfaces. As with any form of communication, human-computer communication must be effective and as efficient as possible. The goals of user interface design are geared toward ensuring that this occurs. To achieve the goals of user interface design, the designer must pay attention to the three elements of the interface—the human, the computer, and the language. Within limits and depending on the application, the user interface designer must then determine what approaches can best accomplish the task while maintaining a “user-friendly” interface. Commensurate with the goals of user interface design, the X Window System is an effective tool for constructing user-friendly interfaces that are also portable across a variety of hardware platforms. Finally, the Ada to X Windows interfaces offer the Ada applications programmer the opportunity to take advantage of the the benefits afforded by the X Window System. The next chapter describes the approach used to develop and integrate the pre-processor with the post-processor as well as the approach used to investigate the feasibility of replacing the current Ada to X Windows interface.

III. Approach

3.1 Introduction

The purpose of this chapter is to describe the approach used to develop and integrate the pre-processor with the post-processor as well as the approach used to investigate the feasibility of replacing the current Ada to X Windows interface.

3.2 Development and Integration of the Processors

A software process model is the "set of tools, methods, and practices to produce a software product"[14]. The primary benefit of a software process model is that it provides "guidance on the order ...in which a project should carry out its major tasks"[1:61]. As noted by Klabunde, although a variety of software process models exist, none is recognized as a standard. With one minor modification, the model chosen for this project was Klabunde's combined methodology depicted in Figure 5.

3.2.1 Combined Methodology. The first phase in the combined methodology is to analyze the problem domain. Basically, this step includes becoming familiar with the terms, concepts, and current state of the problem area. For the purposes of this project, this task included a survey of the development of Saber as well as reviews of the state of the art in user interface design, the X Window System, and X Windows interfaces. The results of the review of the latter three areas was presented in Chapter II.

The second phase of the combined methodology is to develop initial prototypes. The objective here is to clearly identify what is required of the system. We prefer to refer to this step as the requirements specification phase in which prototypes are simply one of the tools available to the analyst. This phase included a thorough review of the development history of Saber to identify the previously documented requirements of the pre-processor. These requirements were then updated as necessary and consolidated into a single document, a requirements specification.

As an aside, given the thesis efforts that preceeded this one [10, 12, 15, 17, 22], the need to generate a requirements specification for the Saber pre-processor was unexpected.

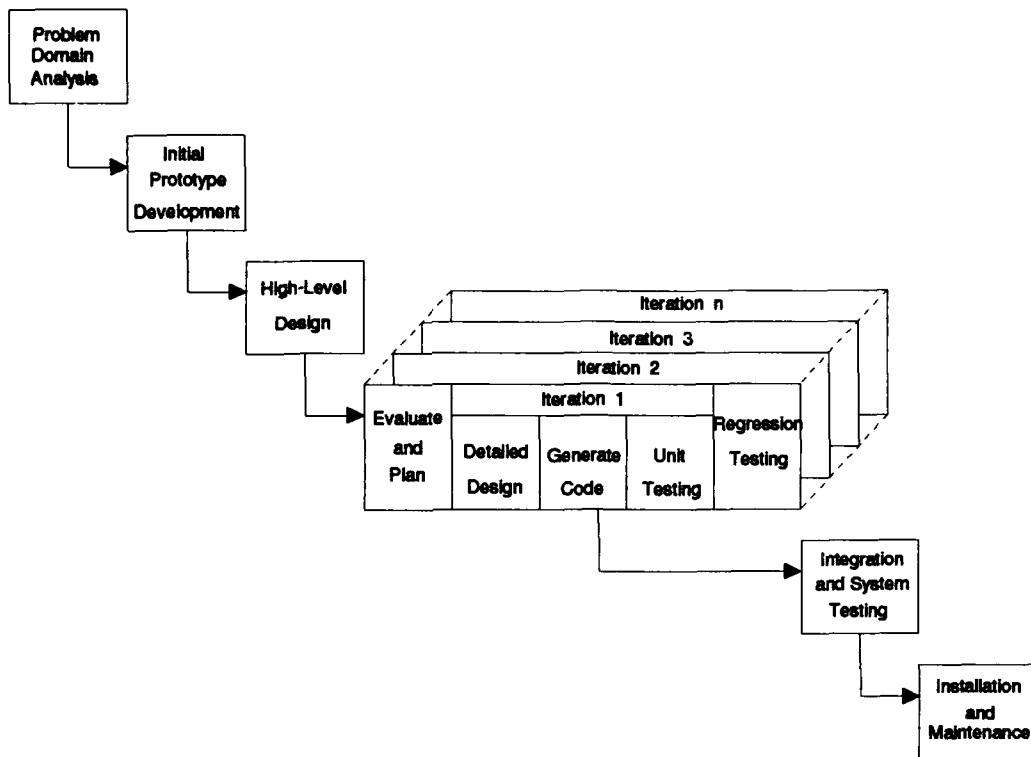


Figure 5. Klabunde's Saber User Interface Design Methodology[12:35]

However, when we attempted to proceed into the detailed design phase of this project, numerous questions about the requirements presented themselves. Even when the answer lay within one of the previous theses, the answer was often difficult to find. This was because no one thesis contained a complete picture of what was required of the pre-processor. Besides being incomplete, there is one other reason requirements specifications need to be updated. Requirements are seldom static and are subject to change. Consequently, the document that describes those requirements is subject to change. Saber is no exception to this. Since vague or unstated requirements often lead to a result other than the one intended, a requirements specification that clearly identifies and consolidates the requirements is essential.

The third phase of the combined methodology is to develop a high-level design. The objective here is to identify the overall structure of the software, or, in other words, identify the major software components and the interfaces between them. Just as there is

no standard software process model, there is no standard high-level design model. However, since object-oriented design is the technique used in the development of Saber to date[10, 12, 22], it seemed only appropriate to apply that technique to this project. Sherry presents a concise summary of the steps required to develop a high-level object-oriented design[22:18]. This phase employed a combination of reverse engineering and forward engineering to integrate the object-oriented designs of Klabunde's post-processor and Horton's pre-processor.

The fourth phase of the combined methodology may be repeated many times and encompasses several mini-phases. In short, this phase implements some component of the high-level design. Two key features of this phase are evaluation and planning and regression testing. Evaluation and planning was used to determine which component of the high-level design should be implemented next and how the component should be integrated with the current system. Regression testing ensured that the new component has not had an adverse effect on a previously implemented component. This was accomplished by retesting the previously implemented components and observing whether the results are still satisfactory.

The fifth phase of the combined methodology is integration and system testing. This phase consisted of the physical integration of Saber's pre-processor and post-processor with the simulation subsystem.

As stated by Klabunde, the sixth and final phase, installation and maintenance, is left to the Air Force Wargaming Center.

3.3 Replacement of the Ada to X Windows Interface

The investigation of the feasibility of replacing the current Ada to X Windows interface, which uses Ada bindings to X Windows, encompassed two steps:

1. Compare the identifiers from the STARS bindings, at least those currently referenced by Saber, with the identifiers from the SERC bindings for equivalence.
2. Assuming the results of the previous step are promising, incrementally replace some of the identifiers from the STARS bindings with identifiers from the SERC bindings.

Inherently, each step included an analysis of the equivalence of the identifiers. The purpose of the evaluation was not only to predict the level of difficulty of conversion but also to analyze potential benefits of converting the identifiers.

The first step was beneficial in several ways and had at least one obvious shortcoming. The benefits of the first step are that it not only provides a complete picture of which identifiers from the bindings are referenced by Saber, but also provides a basis for continuing or discontinuing the investigation. The only apparent shortcoming of the first step is that it had the potential of being very time consuming.

Some may argue that the first step is not required, but there are several reasons not to proceed directly to the second step. First of all, this would assume that it is feasible to replace the STARS bindings with the SERC bindings. However, two facts indicated that the replacement may not be feasible. First, Klabunde pointed out that the STARS bindings, specifically the Boeing bindings, contained some convenience functions which are not a standard part of X Windows, the Xt Intrinsics, or OSF/Motif[12:28]. Consequently, the convenience functions would have to be replaced by implicitly inconvenient functions. Second, some of the SERC bindings, specifically the SERC bindings to the Xt Intrinsics, were not based on the STARS bindings[6]. This too implies that the conversion to SERC's Xt Intrinsics would not be straightforward, if not infeasible. The final reason for not proceeding directly to the second step is that each conversion would implicitly include the actions associated with the first step. Thus, no time savings would be realized by omitting the first step.

3.4 Summary

This chapter discussed the approach used to develop and integrate the pre-processor with the post-processor as well as the approach used to investigate the feasibility of replacing the current Ada to X Windows interface. The approach chosen to develop and integrate the pre-processors is based on a combined software process model described by Klabunde. The approach chosen to investigate the feasibility of replacing the current Ada to X Windows interface is to compare the identifiers from the STARS bindings, at least those currently referenced by Saber, with the identifiers from the SERC bindings and then

incrementally replace some of the identifiers from the STARS bindings with identifiers from the SERC bindings. The next chapter identifies and consolidates the requirements driving the objective of integrating the pre-processor and the post-processor.

IV. Requirements Specification

The purpose of this chapter is to clearly identify and consolidate the requirements driving the objective of integrating the pre-processor and the post-processor. The resulting document is commonly referred to as a requirements specification, and it should contain only a description of *what* is required of a system, not *how* the system should be constructed. More precisely, Zave states that a "requirements specification is supposed to describe all required characteristics of the external behavior of a system, but say nothing about the internal structure that will generate the behavior"[29:102]. Furthermore, it is usually written in layman's terms and supplemented as needed with diagrams or formal notations.

The following discussion presents a concise summary of what is required of the pre-processor. The requirements are derived from previous thesis projects [10, 12, 15, 17, 22] and from discussions with Major Roth[20]. In particular, the requirements identified by Mann and Horton, including Horton's paper prototypes, were valuable sources of information.

4.1 Pre-Processor

The pre-processor should provide a player with the capability to input missions, that is, submit instructions for the assets under his/her control. In effect, the mission input is the stimulus for all events that occur during execution of the simulation. In practice, players specify the missions that are to occur each simulation day, and then the simulation determines the actual outcome of those missions. Mann defined four areas that require mission input[15:178]. Two of the areas are logistical in nature—aircraft beddown missions and supply missions. The other two areas are combative in nature—aircraft missions and land unit missions. Mann describes the concepts associated with each of these mission areas and Horton describes the database entities required to support those concepts [10, 15]. First, the following discussion describes the standard requirements for any mission area, including the input required from a player, the output that should be generated, and the constraints imposed upon a player. Then, the discussion describes each of the four mission areas including the area-specific inputs, outputs, and constraints. Also, to highlight any

changes from previous requirements, the discussion of each mission area will include a summary of the changes to the previous requirements.

4.1.1 Standard Mission Requirements.

4.1.1.1 Inputs. For any mission area, the only standard input requirement is that a player must enter the inputs required for the mission area.

4.1.1.2 Outputs. For any mission area, the standard output requirements are as follows. For each of the required inputs, a player must be able to request help, such as on-screen listings of valid inputs. When a player enters an input, it must be validated. If a player enters an invalid input, the player must be warned and provided help. A player must be able to explicitly accept a mission. When a player accepts a mission, it must be added to a mission list displayed on-screen. The mission list will include not only the missions created during the current scheduling session but also missions created and saved, but not executed, during previous terminal sessions. Potentially, a mission created during a previous scheduling session may contain an input that is no longer valid. If this occurs, the player must be warned at the start of the scheduling session and provided help. A player must be able to edit and delete any mission in the mission list. A player must be able to save the mission list at any time. When a mission list is saved, it should be retained to permit further modification by a player during the current scheduling session.

4.1.1.3 Constraints. For any mission area, the standard constraint requirements are as follows. To be valid, the day and time period requested must be greater than or equal to the current simulation day and time period. To be a privileged player, the player must have successfully requested the privileged side, the WHITE side, at the start of the scheduling session. To be accepted or saved, a mission must meet only one constraint: each of the input values must be valid. Since the pre-processor is required to validate each input, this constraint is ensured implicitly.

4.1.2 Aircraft Beddown Mission Requirements. An aircraft beddown mission deploys aircraft from an airbase, usually but not necessarily a staging base, to another air-

base. The primary sources of information for this requirement are Mann[15:124,179] and Horton[10:49,68].

4.1.2.1 Inputs. For each aircraft beddown mission, several inputs are required. A player must enter the day and time period the mission is to occur, the source airbase, the destination airbase, the type of aircraft and the quantity of aircraft to be deployed.

4.1.2.2 Outputs. In addition to the standard output requirements, a player must be able to execute the mission list. When the mission list is executed, the pre-processor must save the entire mission list and attempt to execute the missions in the mission list. Missions that are successfully executed must be removed from the screen to prevent further modification by a player. Missions that are not successfully executed must be retained on-screen to permit further modification by a player. Also, if an executed mission's source airbase is a staging base, the pre-processor should automatically deploy the minimal supplies needed to service the aircraft, based on the mission's aircraft type. Otherwise, it is incumbent on a player to deploy the minimal supplies through scheduled supply missions. In any case, it is incumbent on a player to deploy additional supplies through scheduled supply missions.

4.1.2.3 Constraints. In addition to the standard constraint requirements, each input must meet the following constraints. The source and destination airbases must be on the same side. Further, they must be on the same side as the player or the player must be a privileged player. Next, the source airbase must have the type of aircraft requested available for the current simulation day and time period, and the destination airbase must have the facilities to service the type of aircraft requested. Finally, the validity of the aircraft quantity depends on the day and time period requested. If the day and time period requested is equal to the current simulation day and time period, then the source airbase must have the quantity of aircraft requested available. If the day and time period requested is greater than the current simulation day and time period, then the quantity of aircraft must simply be some non-negative number. To be executed, a mission must meet two con-

straints. First, as with accepts and saves, each of the input values must be valid. Second, the day and time period requested must be equal to the current simulation day and time period. It is important to note that the pre-processor must be able to execute missions in this mission area. Consequently, the missions executed by the pre-processor are referred to as "immediate" missions. That is, instead of flying the aircraft from the source airbase to the destination airbase, the pre-processor simply decrements the assets available at the source airbase and increments the assets available at the destination airbase immediately.

4.1.2.4 Change Summary. The preceeding requirements contain two significant changes to the requirements outlined by Horton[10:69]. First, the previous requirements did not permit a player to specify the day and time period a beddown mission is to occur. Instead, it was assumed that all beddown missions were for the current simulation day. Unfortunately, this unnecessarily prohibited a player from pre-scheduling missions for future days. Second, the previous requirements did not permit a player to save the mission list at will. This unnecessarily prohibited a player from saving missions periodically as he/she went along. More importantly, this unnecessarily prohibited a player from saving the mission list during one scheduling session and then returning to review/modify the mission list in a subsequent scheduling session. These two changes should make the system more flexible and responsive to a player's needs.

4.1.3 Supply Mission Requirements. A supply mission is a land unit that delivers assets from one unit to one or more other units. The primary sources of information for this requirement are Mann[15:122,180] and Horton[10:39,52,73].

4.1.3.1 Inputs. For each supply mission, several inputs are required. A player must enter the day and time period the mission is to begin, the source unit, the destination unit, the type of supplies to be delivered, the quantity of each supply to be delivered, and whether the supply mission is pre-directed or not.

4.1.3.2 Outputs. In addition to the standard output requirements, a player must be able to execute the mission list. When the mission list is executed, the pre-processor must save the entire mission list and attempt to execute the missions in the

mission list. Missions that are successfully executed must be removed from the screen to prevent further modification by a player. Missions that are not successfully executed must be retained on-screen to permit further modification by a player.

4.1.3.3 Constraints. In addition to the standard constraint requirements, each input must meet the following constraints. The source and destination units must be on the same side. Further, they must be on the same side as the player or the player must be a privileged player. Also, the source unit must be either an airbase depot or a land unit depot. Regardless of what the source unit is, the destination unit can be either an airbase depot, a land unit depot, an airbase, or a land unit. Next, the source unit must have the type of supplies requested available for the current simulation day and time period, and the destination unit must be able to accept the type of supplies requested. Airbases and airbase depots can only accept POL, spare parts, and air weapons. Land units and land unit depots can only accept POL, ammunition, hardware, and land-based weapons. The validity of the supply quantity depends on the day and time period requested. If the day and time period requested is equal to the current simulation day and time period, then the source unit must have the quantity of supplies requested available. If the day and time period requested is greater than the current simulation day and time period, then the quantity of each supply must simply be some non-negative number. To be executed, a mission must meet three constraints. First, as with accepts and saves, each of the input values must be valid. Second, the day and time period requested must be equal to the current simulation day and time period. Third, the player must be a privileged player. It is important to note that the pre-processor must be able to execute missions in this mission area. Consequently, the missions executed by the pre-processor are referred to as "immediate" missions. That is instead of transporting the supplies from the source unit to the destination unit, the pre-processor simply decrements the assets at the source unit and increments the assets available at the destination unit immediately. Immediate missions can be used to simulate airlift of logistics or pre-war positioning of logistics. Finally, except for immediate missions, a supply train must exist that has the tonnage capacity to deliver the supplies requested.

Table 1. Horton's Saber Mission Matrix[10]

Primary Missions	Support Missions					Targets	
	Escort	CAP	SEAD	EC	Refuel	Strike	Area
Offensive Counter Air (OCA)	X		X	X	X	X	
Fighter Sweep (FS)			X	X	X		X
Combat Air Patrol (CAP)			X	X	X		X
Defensive Counter Air (DCA)							X
Air Interdiction (AI)	X		X	X	X	X	
Battlefield Air Interdiction (BAI)	X		X	X	X	X	
Close Air Support (CAS)		X	X	X	X	X	
Reconnaissance (RECCE)	X		X	X	X	X	X
SEAD	X			X	X	X	X
Electronic Combat (EC)	X		X	X	X	X	X
Command and Control (CC)	X		X		X		X
Nuclear (NUKE)	X		X	X	X	X	
Chemical (CHEM)	X		X	X	X	X	X

4.1.3.4 *Change Summary.* No changes are apparent.

4.1.4 *Aircraft Mission Requirements.* An aircraft mission directs aircraft to conduct one of the fundamental mission types defined in Air Force Manual 1-1[7]. Mann defines the aircraft mission types, both primary and support, that need to be portrayed by Saber[15:14,46]. Table 1 depicts the primary mission types that Saber needs to portray. It also depicts the support mission types that may accompany the primary mission types and the possible targets of the primary mission types. The primary sources of information for this requirement are Mann[15:63,117,181] and Horton[10:44,48,73].

4.1.4.1 *Inputs.* For each aircraft mission, a player must enter the day and time period the mission is to reach the target, the priority of the mission, the command headquarters responsible for providing the aircraft, the primary mission type, the rendezvous location for the mission's aircraft, and at least one target.

Also, for each aircraft mission, a player must define the aircraft package to perform the aircraft mission. To do so, a player must enter at least one aircraft type to conduct the primary mission type and may enter one or more aircraft types to conduct support mission types. For each aircraft type, a player must enter the mission type, the aircraft type itself, and the aircraft quantity.

4.1.4.2 *Outputs.* Only the standard output requirements are needed.

4.1.4.3 *Constraints.* In addition to the standard constraint requirements, each input must meet the following constraints. The priority must be some non-negative number. The command headquarters must be on the same side as the player or the player must be a privileged player. The primary mission type must be one of the primary mission types listed in Table 1.

For each aircraft mission, each input for a target must meet the following constraints. Table 1 lists the general valid target categories for each primary mission type. Specifically, an area target must be a geographical location within the theater. On the other hand, a strike target must be a specific object. Horton identified the types of objects that can be strike targets[10:25]. The possible types of objects are land units, airbases, roads, railroads, pipelines, obstacles, cities, and depots. Finally, if the primary mission is CAP or DCA, there can be only one target, the orbit location for the mission.

For each aircraft mission, the aircraft package definition must meet the following constraints. Each mission type must be one of the primary or support mission types listed in Table 1. Furthermore, at least one of the aircraft types in the package must have the same mission type as the primary mission type. Also, if the primary mission type is CAP, no aircraft type can have a mission type of ESCORT. The aircraft type must be available at an airbase within the command headquarter's region of responsibility which is capable of launching combat missions, and must be valid for the mission type. Also, the preferred weapons load for the mission type must be available at the airbase where the aircraft are located. If the day and time period requested is equal to the current simulation day and time period, then the command headquarters must have the aircraft quantity available. If the day and time period requested is greater than the current simulation day and time period, then the aircraft quantity must simply be some non-negative number provided the cumulative quantity of that aircraft type within the command headquarter's region of responsibility is not exceeded. Finally, if the distance from rendezvous location to the target exceeds the radius of any of the aircraft packages, the player must schedule at least one aircraft type of mission type REFUEL.

4.1.4.4 *Change Summary.* The preceeding requirements contain three significant changes to the requirements outlined by Horton[10:69]. First, the previous requirements required a player to enter a return day and time period as well as a duration for orbiting the target for the primary missions DCA and CAP. However, since it is assumed that all aircraft missions will begin and end within a single time period, these three entries are not required. Second, the previous requirements required a player to enter an orbit location for the primary missions DCA and CAP. However, for these two mission types, the orbit location *is* the target. Thus, the return day and time period and orbit location are unnecessary. Third, the previous requirements required a player to enter a mission class, either CONVENTIONAL, NUCLEAR, or BIOLOGICAL/CHEMICAL, for each aircraft mission. However, two of the primary mission types are Nuclear and Chemical, which implies the remaining primary mission types are conventional. Thus, the mission class is redundant and is not required.

4.1.5 *Land Unit Mission Requirements.* A land unit mission models one of the three general categories of army operations: offensive, defensive, or retrograde. Saber is required to model the following army mission types: attack, defend, withdraw, movement, and support[22:65]. The primary sources of information for this requirement are Ness[17:24,37-41], Mann[15:181], and Horton[10:49,52,72].

4.1.5.1 *Inputs.* For each land unit mission, a player must enter the day and time period the mission is to begin, the land unit to perform the mission, the mission type, the target, and an indicator of whether the mission should override previous instructions to the land unit.

4.1.5.2 *Outputs.* Only the standard output requirements are needed.

4.1.5.3 *Constraints.* In addition to the standard constraint requirements, each input must meet the following constraints. The land unit must be on the same side as the player or the player must be a privileged player. The mission type must be one of the mission types listed above. The target must be a geographical location within the theater

or a land unit, airbase, or depot. If the mission type is *support*, the target must be on the same side as the land unit performing the mission.

4.1.5.4 *Change Summary.* No changes are apparent.

4.2 *Summary*

The chapter clearly identifies and consolidates the requirements driving the objective of integrating the pre-processor and the post-processor. This effort resulted in a complete and current requirements specification of the four areas of the pre-processor that require mission input. The next chapter describes the integrated design of the pre-processor and the post-processor.

V. *Integrated Design of the Processors*

5.1 *Introduction*

The purpose of this chapter is to describe the integrated design of the pre-processor and the post-processor. This phase will employ a combination of reverse engineering and forward engineering to integrate the object-oriented designs of the two processors. As stated earlier, object-oriented design is the model chosen for the development of this project. Informally, object-oriented design decomposes a system into “a collection of discrete objects that incorporate both data structure and behavior” [21:1].

Just as there is no *standard* design model, there is no *standard* object-oriented design model. However, previous Saber projects[12, 22] have met success using a model based on the one described by Booch[2, 3]. Therefore, it seemed only natural to use Booch’s model as the basis for development of this project. Booch’s object-oriented development model includes the following steps[3:17]:

1. *Identify the objects and their attributes.* A practical approach for doing this is to simply extract the nouns from the requirements specification and refine them to produce the objects and their attributes.
2. *Identify the operations suffered by and required of each object.* A practical approach for doing this is to simply extract the verbs from the requirements specification and refine them to produce the operations on the objects.
3. *Establish the visibility of each object in relation to other objects.* A practical approach for doing this is to identify which objects need to communicate with one another.
4. *Establish the interface of each object.* This is the physical implementation of the previous step. Since Ada is the “language of choice” for this project, development of Ada specifications satisfies this step.
5. *Implement each object.* This is the physical implementation of objects, including their attributes and operations. Similar to the previous step, development of Ada bodies satisfies this step.

Sherry[22:6,18] describes each of these steps in detail. Therefore, they will not be explained further here. However, it should be noted that the first three steps of Booch's model are especially applicable during the design phase; whereas, the last two steps are applicable during the development phase.

5.2 *Engineering the Design*

First, it is important to note that the software development sequence used thus far (requirements specification, design, and then implementation) is commonly referred to as forward engineering. "Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system"[5:14]. Klabunde[12] and Horton[10] also used this sequence. In doing so, Klabunde produced a well-documented design and implementation of the Saber post-processor. Similarly, Horton produced a fairly well-documented detailed design of the Saber pre-processor, but the high-level design was either non-existent or not explicitly stated. Also, Horton implemented part of the pre-processor.

Consequently, the integrated design of the processors is based on the following sources of information:

- *The pre-processor requirements specification stated in this document.* This is a factor since the pre-processor design is based on the pre-processor requirements specification.
- *The pre-processor design described by Horton.* One might argue that this is not a factor because of the previous one. However, requirements specifications are seldom complete. Since Horton has partially implemented the pre-processor, he may have identified some objects and interfaces that are not apparent in the pre-processor requirements specification.
- *The post-processor design described by Klabunde.* This is a factor since the integrated design of the processors is a composition of the post-processor design and the pre-processor design. Since Klabunde had almost completed the post-processor

implementation, he too may have identified some objects and interfaces that are not reflected in the post-processor design.

5.2.1 Reverse Engineering the Existing Designs. Since the existing implementations of the processors are a physical reflection of their conceptual design, it would seem worthwhile to extract the design from the implementations. Not only would this provide insight into what the components of the integrated design should be, but also the resulting information could be used in conjunction with the existing design documentation to provide the the foundation for an integrated design. The technique for doing this is called reverse engineering. Chikofsky defines reverse engineering as follows[5:15]:

Reverse engineering is the process of analyzing a subject system to

- identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction.

Furthermore, the main purpose of reverse engineering is to "increase the overall comprehensibility of the system for both maintenance and new development"[5:16]. Thus, reverse engineering seems to be ideally suited for deriving the existing design, including the objects and their interfaces.

The actual task of reverse engineering the existing pre-processor and post-processor proved to be quite straightforward. The first step taken was to identify the objects, including their attributes and operations. This began with the identification of library level Ada modules of the pre-processor and post-processor. Assuming they were created in a manner consistent with Booch's model, these modules should correspond to some high-level object class. As the reader will recall, the post-processor's object-oriented design, including its object classes was well documented. Therefore, its object classes were used as a template for classifying the implementation modules. This proved to be quite successful. However, some of the modules did not logically fit any of the post-processor design classes. After careful review of the structure and purpose of these remaining modules, they were composed into their logical classes.

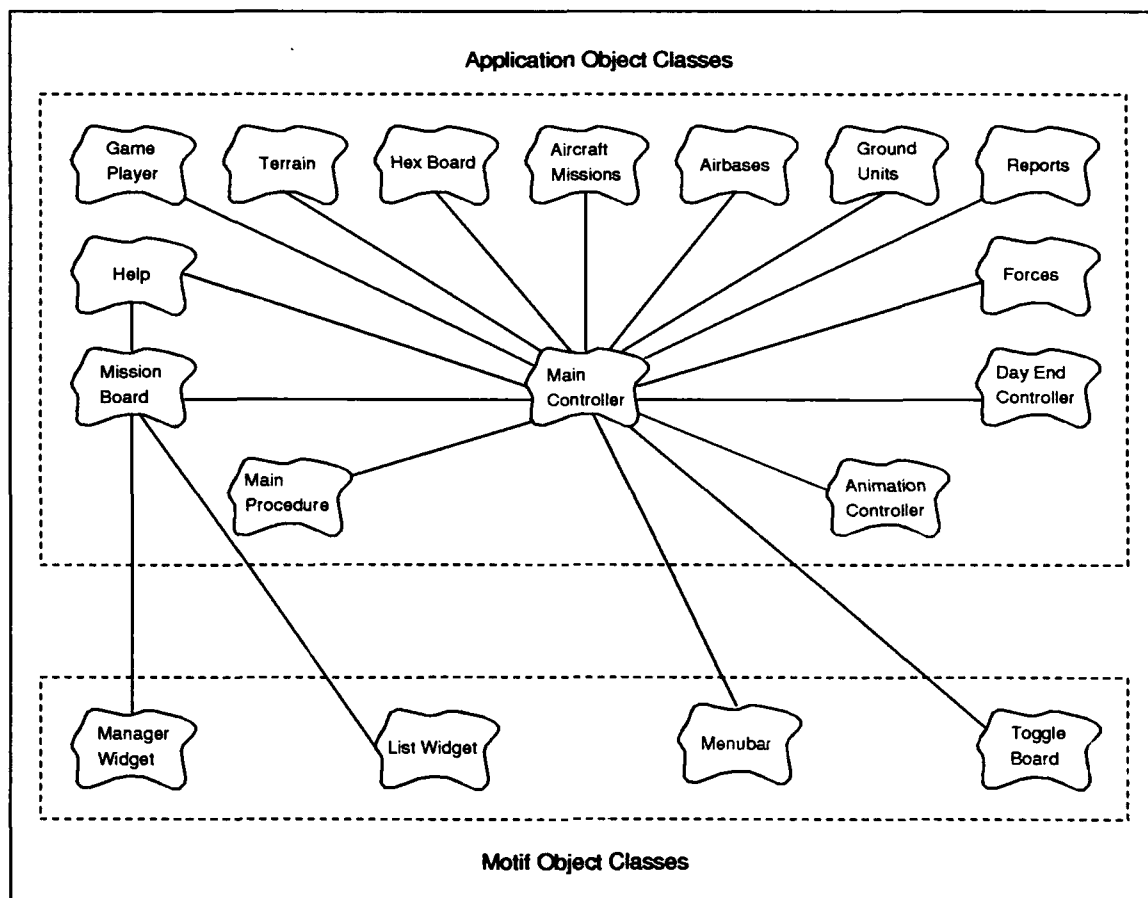


Figure 6. Integrated User Interface Object Diagram

The next step was to identify the interfaces between the objects. This began with a review of the Ada library level modules to determine what other library level modules they referenced. Fortunately, Ada requires the developer to use context clauses to reference library modules. A context clause is contained within the calling module and must specify the called library module by name. Consequently, the task of identifying the interfaces between the objects was fairly simple. However, one problem was encountered. Some calling modules contained extraneous context clauses. In other words, the calling module never actually invoked any identifier within the called module. At first, these extraneous context clauses led us to believe certain interfaces existed when, in fact, they did not. These extraneous context clauses were eliminated from the modules. Figure 6 is an updated version of Klabunde's object diagram[12:48] and reflects the object classes and the interfaces between them that were derived through this reverse engineering process.

5.2.2 Forward Engineering the Integrated Design. As mentioned earlier, the integrated design of the pre-processor and post-processor is based on three sources of information: the pre-processor requirements specification, the pre-processor design by Horton, and the post-processor design by Klabunde. Fortunately, the information needed from Horton and Klabunde was available from their theses[10, 12] and from the reverse engineering process just described. The two remaining design tasks were to create a high-level design from the pre-processor requirements specification following the steps outlined by Booch[3:17] and to integrate it with the existing high-level design. As hoped, these steps had minimal impact on the existing high-level design since many of the pre-processor's requirements were already reflected in it. Only two changes of significance were required to integrate the processor designs. First, some classes existed in both processors that were redundant. These were merged into a single class. Second, a couple of the classes were actually the composition of two or more disparate classes. These were decomposed into separate classes. These changes were incorporated into Figure 6.

The classes fall into two distinct categories: application object classes, and Motif object classes. Perhaps the most distinguishing feature between these two categories is reusability. The application object classes are unique to Saber, and with some tailoring, may be reusable elsewhere within the wargaming problem domain. On the other hand, the Motif object classes are neither application unique nor domain unique. Thus, with little or no tailoring, they may be reusable across a variety of problem domains. While Appendix A describes each object class in detail, a summary of each object class is presented below:

- Application Object Classes
 - Airbases: This class has methods for creating, displaying, and erasing airbase symbols; and for displaying and erasing airbase status boards.
 - Aircraft Missions: This class has methods for creating, displaying, and erasing aircraft mission symbols; and for displaying and erasing aircraft mission status boards.
 - Animation Controller: This class instantiates objects throughout the previous simulation day.

- Day End Controller: This class instantiates objects representing the state of the simulation at the end of the previous simulation day.
- Forces: This class has methods for reading the force data file and retrieving the force identifier (side) of a given country.
- Game Player: This class has methods for setting and retrieving the player's side, the seminar number, and the current simulation day.
- Ground Units: This class has methods for creating, displaying, and erasing ground unit symbols; and for displaying and erasing ground unit status boards.
- Help: This class has methods for creating the information needed for context sensitive help.
- Hexboard: This class has methods for creating and deleting hexboards.
- Main Controller: This class instantiates objects that are common to the Animation Controller and the Day End Controller.
- Main Procedure: This class displays the simulation startup form and instantiates the simulation startup task that processes X Windows events.
- Mission Board: This class has methods for creating, displaying, and erasing mission input boards.
- Reports: This class has methods for specifying a standard set of reports, printing reports, and viewing reports.
- Terrain: This class has methods for reading, displaying, and erasing terrain symbols. The terrain symbols denote trafficability, roads, rivers, railroads, obstacles, Forward Edge of Battle Areas (FEBA's), country borders, coast lines, pipelines and cities.

- Motif Object Classes

- List Widget: This class has methods for creating scrolled lists; adding, deleting, and replacing items in the scrolled lists.
- Manager Widget: This class has methods for creating subclasses of the manager widget class including dialogs, labels, and pushbuttons.

- Menubar: This class has methods for the creating menubars and adding pull-down menus.
- Toggle Board: This class has methods for creating and deleting bulletin boards that contain toggle buttons. It also has methods for creating and deleting data structures that are passed as parameters for the board creation.

It should be noted that four of the object classes (the driver procedure, the main controller, the animation controller, and the day end controller) are not true application object classes, since they are not from the problem domain. However, since they are essential elements of the integrated design, they are included here to provide a complete picture.

5.3 Summary

This chapter described the integrated object-oriented design of the pre-processor and the post-processor. This effort employed a combination of reverse engineering and forward engineering to produce the integrated object-oriented design. The next chapter discusses the integrated implementation of the pre-processor and post-processor as well as an analysis of the feasibility of replacing the current Ada to X Windows interface.

VI. Implementation and Investigation

6.1 Introduction

The purpose of this chapter is to describe the integrated implementation of the processors and to present an investigation of the feasibility of replacing the current Ada to X Windows interface. The approach outlined in Chapter III depicts the act of integration as one of the last phases of the software development process. However, as implied by the integrated design phase in Chapter V, integration was a consideration throughout the software development process. As a result, the actual act of integration proved to be fairly straightforward. Therefore, the initial focus of the following discussion will be the integrated implementation of the processors. Next, since Klabunde presents a thorough description of the the post-processor's[12] implementation, this discussion will describe the implementation of only those objects integral to the pre-processor. Finally, an investigation of the feasibility of replacing the current Ada to X Windows interface is presented.

6.2 Integrated Implementation of the Processors

Figure 7 is an updated version of Klabunde's module diagram[12:83]. It depicts the Ada library level modules and the interfaces that implement the integrated design of the processors. The modules correspond to the object classes, and the interfaces between modules correspond to the interfaces between object classes. The particular Ada modules chosen to implement the object classes were packages and subprograms. Conceptually, a module at the head of an arrow depends on the module at the tail of the arrow. In the physical implementation, the Ada module at the head of an arrow uses an Ada context clause to provide visibility to the module at the tail of the arrow.

The Ada modules function according to the object class descriptions in Chapter V. As a reminder, descriptions of the modules that *drive* the simulation are repeated here. The Main Procedure displays the simulation startup form and instantiates the simulation startup task that processes X Windows events. The Main Controller instantiates objects that are common to the Animation Controller and the Day End Controller. The Animation Controller instantiates objects throughout the previous simulation day. Finally, the Day

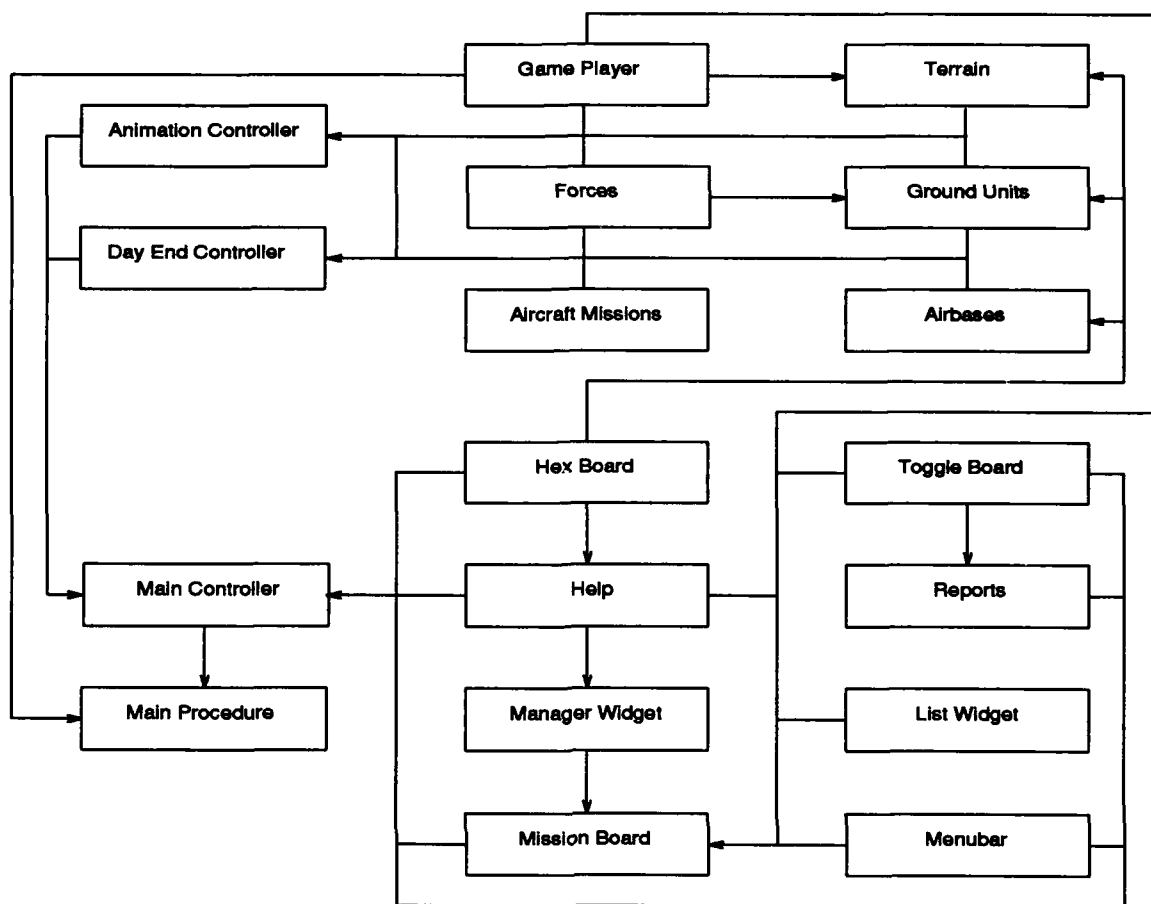


Figure 7. Integrated User Interface Ada Module Diagram

End Controller instantiates objects representing the state of the simulation at the end of the previous simulation day.

6.3 Integrated Implementation of the Pre-Processor

As noted earlier, Horton had implemented a subset of the pre-processor. Unfortunately, several factors detracted from its readability and maintainability. Therefore, before continuing with the implementation of the pre-processor, the following preliminary steps were taken:

- *The Verdex source code formatter was used to improve the readability of the Ada modules.* This step, in conjunction with manual intervention, made the code indentation and identifier capitalization consistent.

- *Extraneous context clauses like those described in Chapter V were removed from the Ada modules.* This eliminated unnecessary interfaces with other modules. This also caused minor reductions in the memory consumption of the user interface executable (8 kilobytes) and the Ada development library (39 kilobytes).
- *All imported identifiers were qualified (prefixed) with the name of the Ada library module that declared them.* This eliminated an undesirable side effect associated with the Ada *use* clause. That clause eliminates the constraint that requires developers to prefix imported identifiers with the imported module name. When used for this purpose, it “can cause a loss of clarity and could introduce a name clash”[3:222]. In fact, this problem did exist in the pre-processor.
- *Commensurate with the reverse engineering task in Chapter V, the Ada modules were recomposed consistent with the classes they represented.*
- *As needed, the Ada identifiers, such as type, variable, procedure and function names, were renamed to give them meaningful names.* The Ada Quality and Style guide[23] proved to be an excellent source of naming conventions.

Once these preliminary actions were taken, the development of the pre-processor continued in earnest. Of the object classes identified during the integrated design phase, several are integral, if not exclusive, elements of the pre-processor. They are the Help, List Widget, Manager Widget, and Mission Board object classes. Figure 7 depicts the Ada modules that implement these object classes. The following discussion describes these Ada modules and the OSF/Motif structures used to implement the object classes.

6.3.1 Help. The Help module is implemented as an Ada package and contains static, predefined message strings as well as subprograms for dynamically creating the information needed for context sensitive help. Figure 8 is an example is an instantiation of a Motif information dialog containing one of the Help module’s predefined messages. The predefined messages reflect several of the characteristics of good user interface design, including a simple and natural dialog, appropriate feedback, and suggestions of what to do next. On the other hand, the Help module’s subprograms can dynamically generate help based on the current situation. For example, a subprogram in the Help module generated

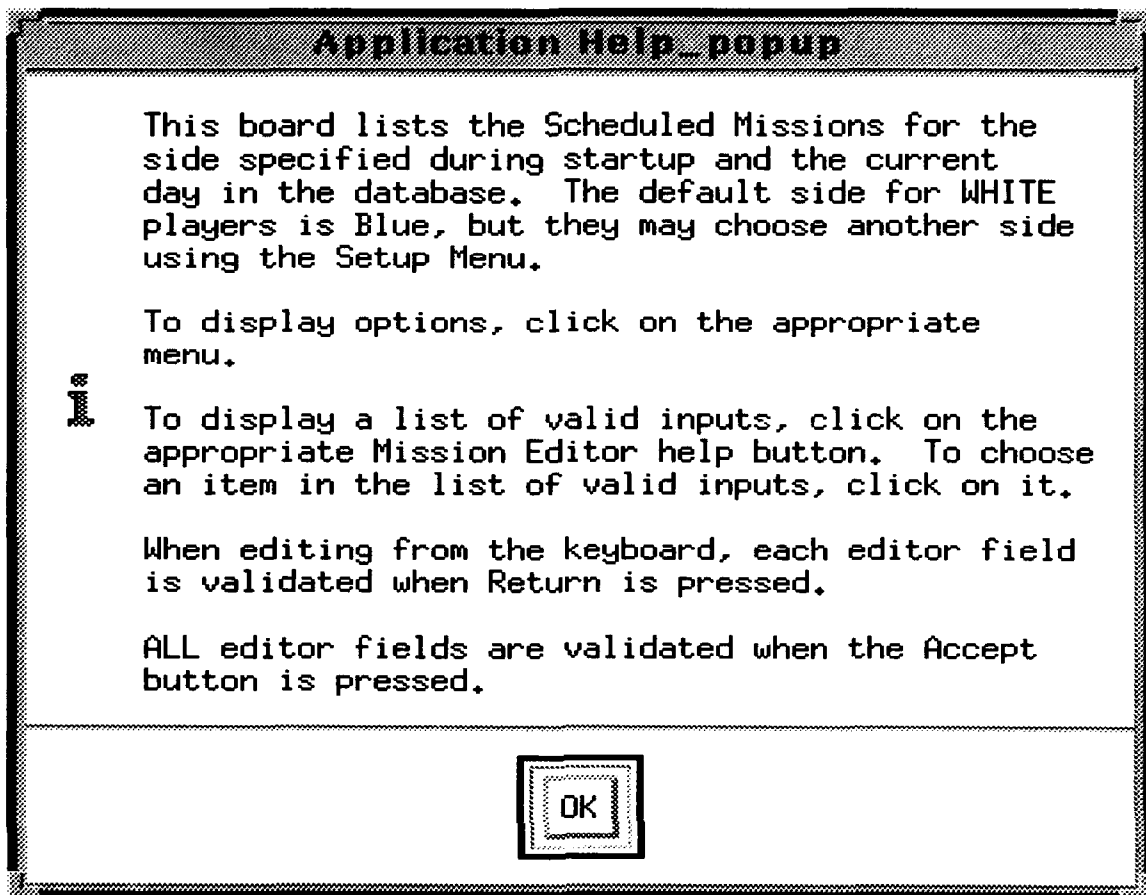


Figure 8. Beddown Mission Board Help

the information displayed in Figure 9. That subprogram generated a list of the valid aircraft types based on the source and destination airbases that were entered by a player. The Help module contains subprograms to generate context sensitive help information for each Mission Board data entry field.

6.3.2 List Widget. Figure 9 also includes an instantiation of an Motif scrolled list. The List Widget module is implemented as an Ada package and contains subprograms for creating scrolled lists as well as subprograms for adding, deleting, and replacing items in the scrolled lists. List Widget defines the structure and content of a given instance of a scrolled list upon instantiation. Consequently, the List Widget subprograms are designed to operate on virtually any Motif scrolled list.

Aircraft Type Help Dialog .po		
M17A	FRESCO	37
M21A	FISHBED	2
M23A	FLOGGER	3
SU7A	FITTER	21
SU17A	FITTER C	18
SU24A	FENCER	41
SU25A	FROGFOOT	13
AN22C	COCK	6

Figure 9. Aircraft Type Help

6.3.3 Manager Widget. The Manager Widget module is implemented as an Ada package and contains subprograms for creating a variety of the Motif manager widget class including dialogs, labels, and pushbuttons. Figures 10, 11, and 12 are examples of three Motif convenience dialogs created by Manager Widget. The convenience dialogs demonstrate several important features. For example, to get the user's attention, the titles of the error and warning dialogs are capitalized. Also, the messages (predefined in the Help module) in all three dialogs describe the current situation precisely and suggest a course of action. Finally, warning messages are typically associated with actions that will lead to a change in state such that the previous state is irrecoverable. Therefore, they include a Cancel button to prevent the change in state.

6.3.4 Mission Board. The Mission Board modules are implemented as Ada packages and contain subprograms for creating, displaying, and erasing the four subclasses of the Mission Board object class in the Aircraft Beddown Mission Board, the Supply Mission Board, the Aircraft Mission Board, and the Land Unit Mission Board. These four subclasses represent the core of the pre-processor and are relatively active compared to the three object classes discussed thus far. The Mission Board object class causes the instantiation of the other object classes—Help, List Widget, and Manager Widget. Each

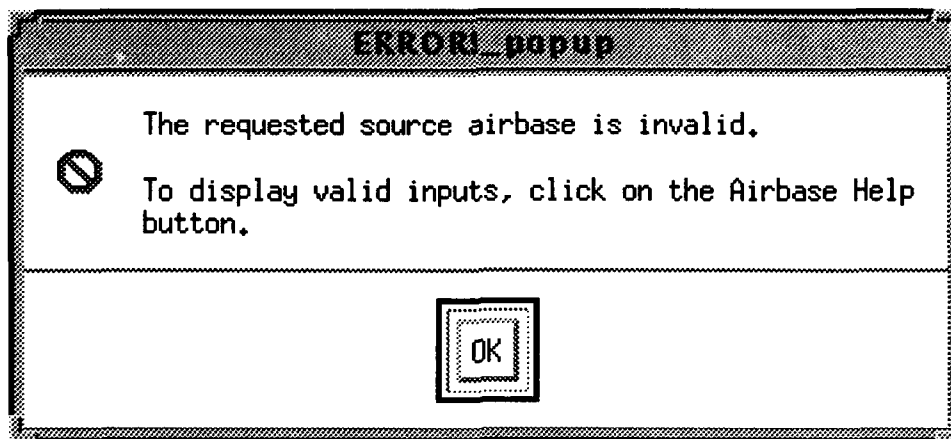


Figure 10. Source Airbase Error Dialog

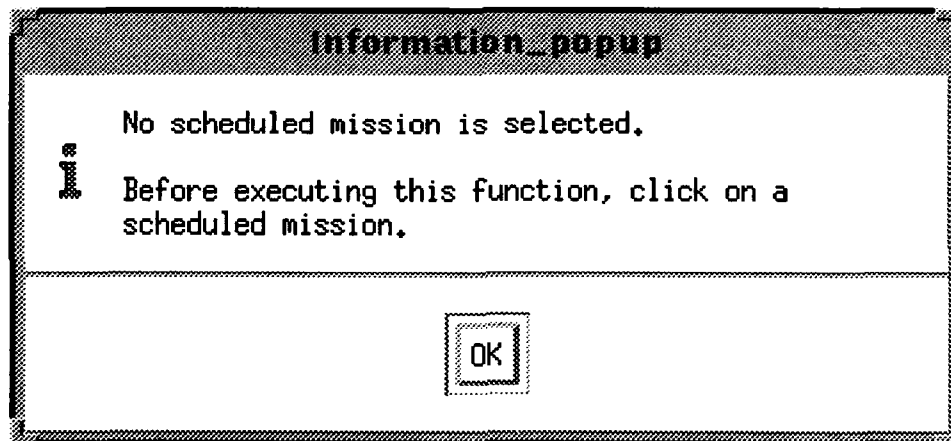


Figure 11. Select a Mission Information Dialog

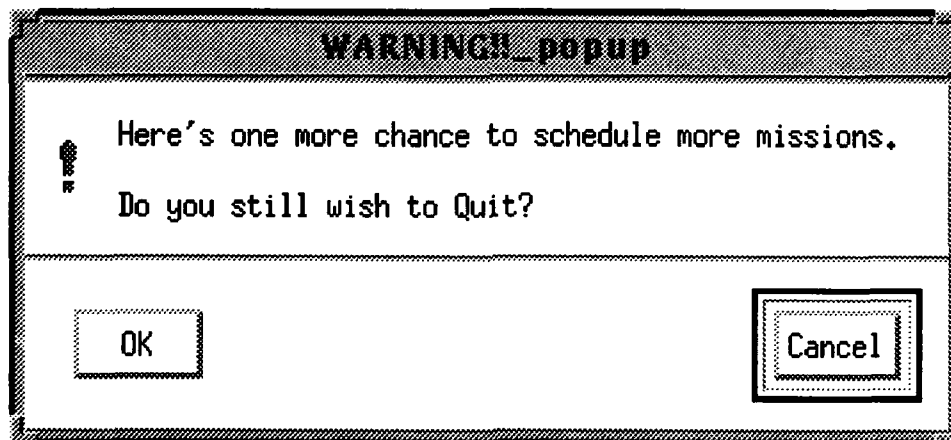


Figure 12. Quit Mission Board Warning Dialog

mission board contains a similar composition of Motif structures. Likewise, though the specific purpose of each mission board differs, the functionality available with each board is similar. Therefore, the following discussion describes only one of the mission boards, the Aircraft Beddown Mission Board.

6.3.4.1 Aircraft Beddown Mission Board. Figure 13 depicts an instance of the Aircraft Beddown Mission Board with some sample data entries. Although a bulletin board dialog provides the foundation for the mission board, a main window provides the template for structuring its components. In this case, the main window provides the menubar and a work area beneath it. Since the work area needs to support two functionally distinct purposes, it contains two forms—one to provide a template for the Scheduled Missions and one for the Mission Editor. The Scheduled Missions form contains a scrolled list widget to display the scheduled missions and a variety of manager widgets such as labels and pushbuttons. Similarly, the Mission Editor contains a variety of manager widgets including text widgets for data entry. Notice that one of the buttons, the Cancel button, in Figure 13 is “greyed” out. It is an invalid option at this time. Similarly, if there were no scheduled missions in Figure 13, all of the buttons in the Scheduled Missions form would be “greyed” out. This feature avoids the delay and aggravation associated with permitting a player to select an invalid option and then informing the player that it was an invalid option.

When the mission board is created and displayed during a given terminal session, the list of scheduled missions may contain missions created and saved during previous terminal sessions. If so, these missions are still subject to revision. The default side for privileged players is Blue, but they may choose another side using the Setup menu. The default side for nonprivileged players is the side they selected during the start of the terminal session. The day and time period displayed is determined by the database clock.

To schedule a mission, a player must enter valid data into each of the four text entry fields and click on the Accept button. For the source and destination airbases, a player may enter either the airbase number or the airbase name. Likewise, for the aircraft type, a player may enter either an aircraft identifier or name. If the Return key

Aircraft Beddown Mission Board_popup

Side: Blue
 Scheduled Missions
Day/Period: 1/ 3

Source Airbase	Destination Airbase	Aircraft Type	Quantity	
AB000003	CHONG_JU	AB000004	CHUKPYON	F4E PHANTOM II 12
AB000003	CHONG_JU	AB000004	CHUKPYON	F5A TIGER II 10

Mission Editor

Source Airbase	Destination Airbase	Aircraft Type	Quantity	
<input type="text" value="AB000008"/>	<input type="text" value="AB000007"/>	<input type="text" value="F5A"/>	<input type="text" value="10"/>	<input type="button" value="Accept"/>
<input type="button" value="Airbase Help"/>		<input type="button" value="Aircraft Help"/>		<input type="button" value="Cancel"/>

Figure 13. Beddown Mission Board

is pressed after data is entered into a field, the data is immediately validated. Otherwise, the data is not validated until the player clicks on the Accept button. To help compensate for typographical errors by the player, the validation subprograms strip out any non-alphanumeric characters, convert alphabetic characters to upper case, and then compare the data with the simulation's database. If the data is valid, it is simply echoed into the the input field. Otherwise, an error message is generated like the one in Figure 10. For each field or group of fields, a player may request a list of valid inputs, by clicking on the appropriate Mission Editor help button. Clicking on the Airbase Help button generates a list of valid airbases like the one in Figure 14.

Rather than entering data into a text field from the keyboard, a player may choose data from the list of valid inputs by clicking on an item in the list. The item is then

Airbase Help Dialog...popup		
AB000059	YONPO	YONPO
AB000058	WONGYO	WONGO RI HWY STRIP
AB000057	TOKSAN	TOKSAN
AB000056	TATONG	TA TONG KOU
AB000011	ONYANG	ONYANG HWY STRIP
AB000010	NAEGI	NAEGI RI HWY STRIP
AB000009	KWANG_JU	KWANG JU
AB000008	KUNSAN	KUNSAN
AB000007	KIMHAE	KIMHAE
AB000006	KIMCHON	KIMCHON HWY STRIP
AB000005	KANGNUNG	KANGNUNG
AB000004	CHUKPYON	CHUKPYON HWY STRIP
AB000003	CHONG_JU	CHONG JU
AB000002	CHONGUP	CHONGUP HWY STRIP
AB000001	CHEJU	CHEJU INTERNATIONAL AIRPORT

Figure 14. Airbase Help

echoed in the text field. In the case of the Airbase Help list, the first click echoes the data in the source airbase text field, and the second click echoes the data in destination airbase field. Thereafter, the system toggles back and forth between the two fields. Clicking on the Aircraft Help button generates a list of valid aircraft types like the one in Figure 9. Once a player chooses an aircraft type, a list of valid aircraft quantities like the one in Figure 15 is generated for the player.

Once a player accepts a mission by clicking on the Accept button, the mission is added to the list of Scheduled Missions. A player may Edit or Delete any mission in the mission list. To select the mission to be edited or deleted, the player must click on the desired mission and then click on the Edit or the Delete button, respectively. If a player Edits a mission, it is echoed in the text fields of the Mission Editor. The player may then revise the mission and Accept the revision or Cancel the edit. If a player attempts to Delete a mission, a warning is displayed like the one in Figure 16.

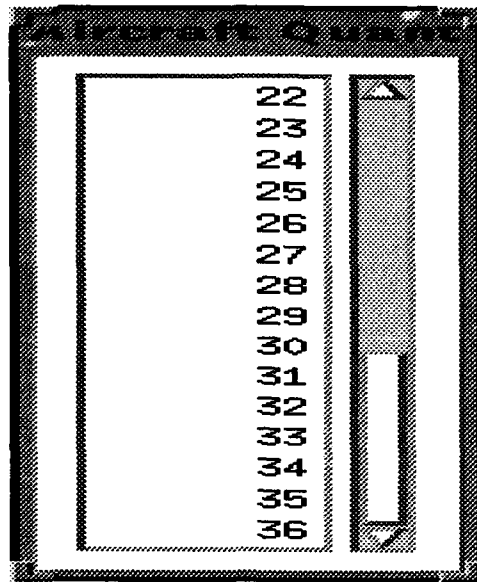


Figure 15. Aircraft Quantity Help

Provided there is at least one mission in the list of scheduled missions, a player may Save or Execute them. As noted in Figure 17, a player may Save the mission list to the database at will and continue editing missions and adding them to the mission list. On the other hand, as noted in Figure 18, the purpose of the Execute button is twofold. It first saves the current list of scheduled missions to the database, and then it executes those missions. Finally, to prevent additional editing of the executed missions, it removes them from the scheduled missions list.

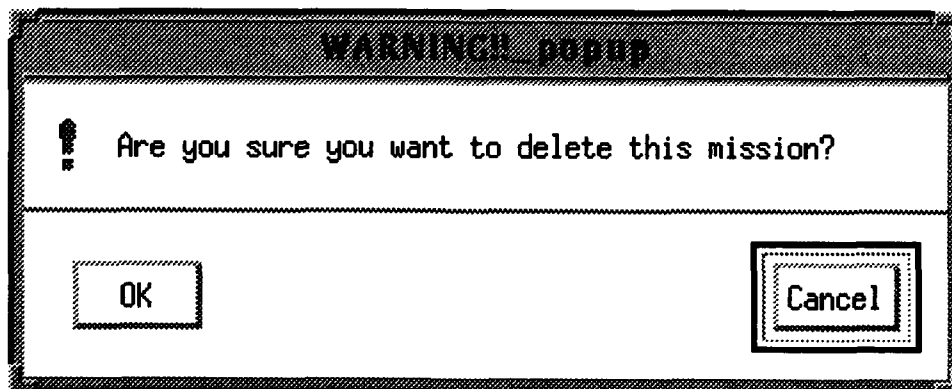


Figure 16. Delete Mission Warning Dialog

Finally, to end the terminal session, a player must click on the Quit menu. The player must then Confirm or Cancel the request to quit. If the player Confirms the request to Quit and there are any unsaved missions in the scheduled missions list, a warning is displayed like the one in Figure 19. Since the scheduled missions list is only saved at the request of a player, this helps to avoid those occasions when the player may forget to save it.

6.4 Investigation of Replacing the Ada to X Windows Interface

As described in Chapter III, a two-phased approach was taken to investigate the feasibility of replacing the current Ada to X Windows interface:

1. Compare the identifiers from the STARS bindings, at least those currently referenced by Saber, with the identifiers from the SERC bindings for equivalence.
2. Assuming the results of the previous phase are promising, the next logical phase would be to incrementally replace some of the identifiers from the STARS bindings with identifiers from the SERC bindings.

The following discussion describes the steps taken during each phase of the investigation. However, we first need to define some terminology and refine the scope of the investigation. Strictly speaking, an identifier can be any Ada word. However, the term is usually used to refer to the name of some Ada entity, such as a variable, constant, data type, procedure, or function. Next, *visible* identifiers are those that are declared in an Ada library level module and can be imported (referenced) by another Ada module. Generally speaking, any separately compilable Ada program unit can be a library level module, provided it conforms to Ada's visibility rules. The four types of Ada program units are subprograms, packages, tasks, and generic units[3:55].

The scope of this investigation could have included a comparison of all of the visible STARS identifiers with all of the visible SERC identifiers. However, since our immediate concern was to investigate the feasibility of replacing the Ada to X Windows interface currently used by the Saber user interface, the scope was narrowed to include only those

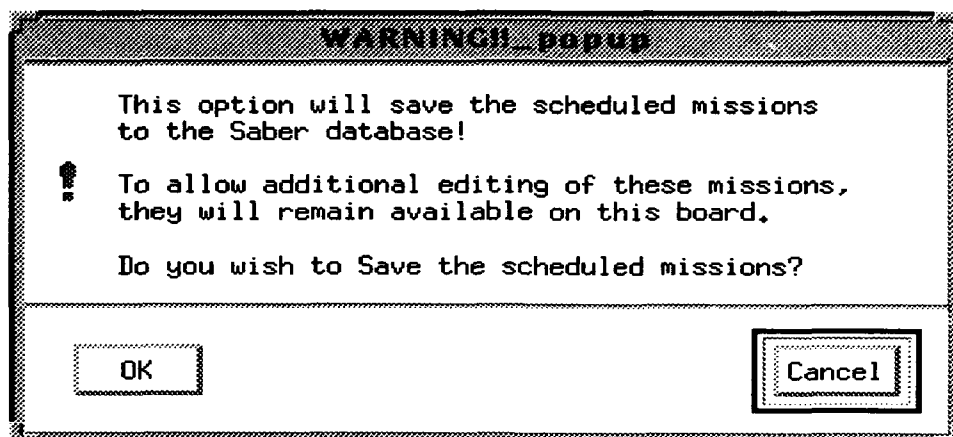


Figure 17. Save Missions Warning Dialog

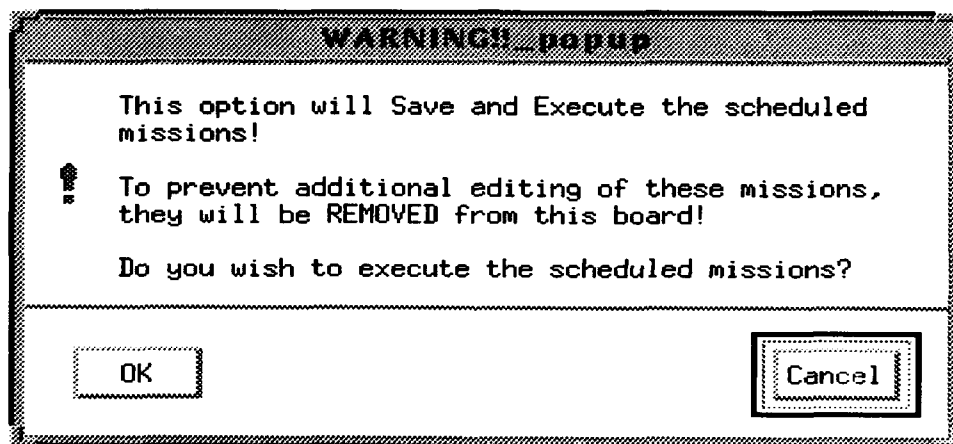


Figure 18. Execute Missions Warning Dialog

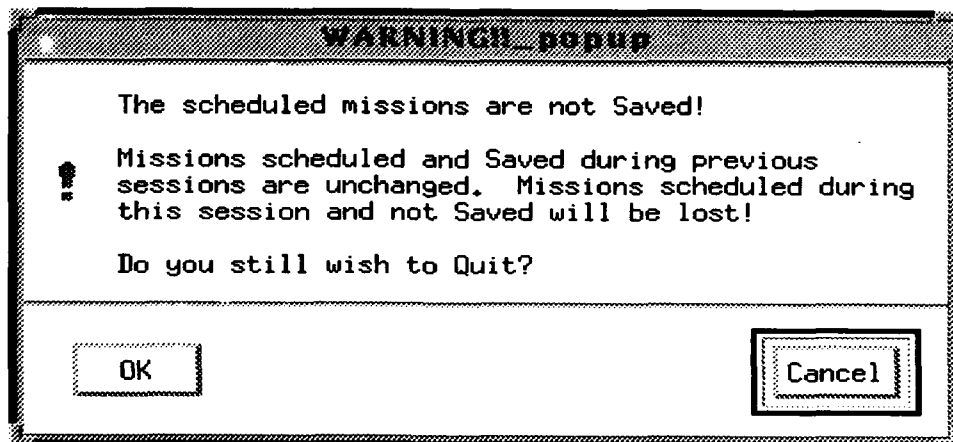


Figure 19. Quit with Unsaved Missions Warning Dialog

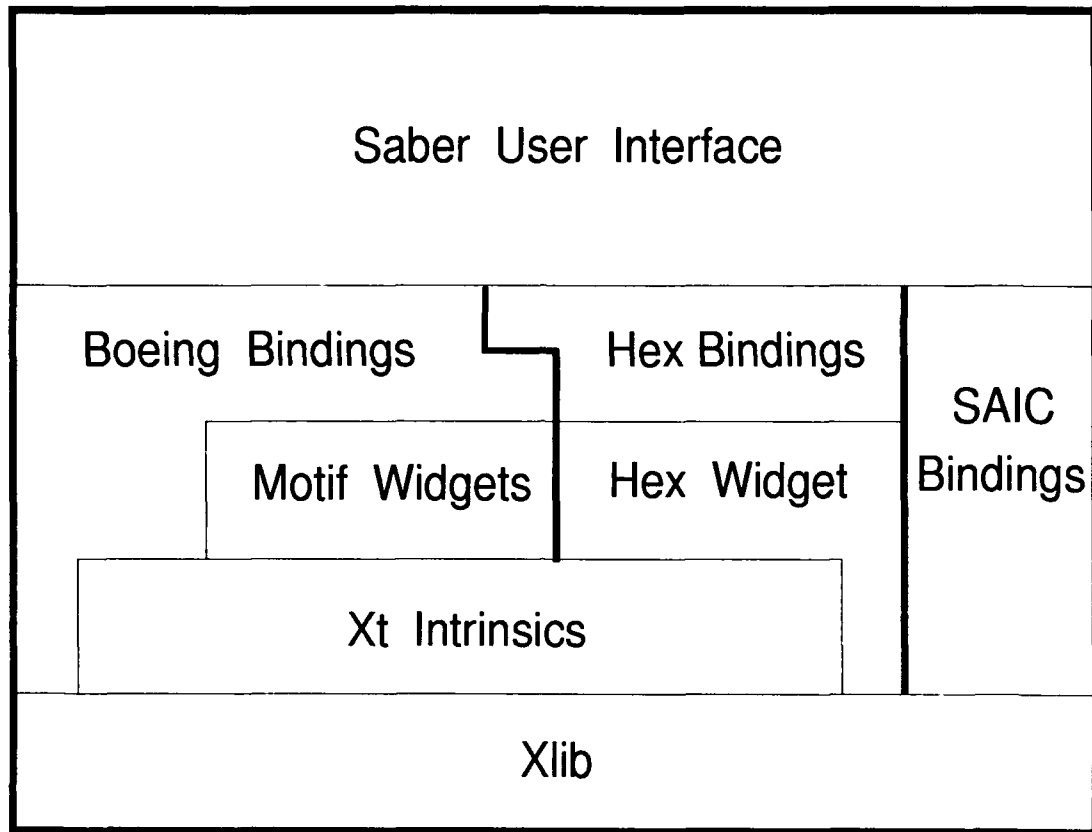


Figure 20. User Interface Relationship to the Ada Bindings

STARS identifiers imported by it. Figure 20 is a revision of a figure by Klabunde[12:53]. It depicts the relationship of the Saber user interface to all of the Ada bindings, including the STARS bindings developed by Boeing and SAIC. Each layer in Figure 20 has visibility to the layer directly beneath it. Thus, a given layer can import the identifiers declared in any layer visible to it. During the course of the research, we discovered that the Hex bindings also import identifiers from the Boeing bindings. Consequently, the scope of the comparison was broadened to include any STARS identifiers imported by either the Saber user interface or the Hex bindings.

6.4.1 Comparison of the STARS and SERC Identifiers. Before we could compare the identifiers, we first had to determine which ones were imported by the Saber user interface and the Hex bindings. This task required several steps itself:

1. *Identify the library level STARS modules.* To do this, we searched the source code of the Boeing and SAIC bindings for instances of library level program units. We found that the Boeing bindings contained five packages that were library level modules: AFS_Basic_Types, Motif_Resource_Manager, Xlib, XM_Widget_Set, and X_Toolkit_Intrinsics_OSF. We found that the SAIC bindings contained four packages that were library level modules: Command_Line_Arguments, Key_Syms, X_Windows, and X_Windows_Interface.
2. *Identify which library level STARS modules are imported by the Saber user interface and the Hex bindings.* To do this, we searched the source code of the Saber user interface and the Hex bindings for context clauses containing the binding package names. Since extraneous context clauses had been removed during another phase of this thesis project, this would provide an accurate list of the imported modules. The Saber user interface imported four of the Boeing packages (AFS_Basic_Types, Xlib, XM_Widget_Set, and X_Toolkit_Intrinsics_OSF) and one of the SAIC packages (X_Windows). The Hex bindings imported three of the Boeing packages (AFS_Basic_Types, Xlib, and X_Toolkit_Intrinsics_OSF) and none of the SAIC packages.
3. *Identify which identifiers are imported from the STARS modules by the Saber user interface and the Hex bindings.* To do this, we searched the source code of the Saber user interface for the name of each imported module. Since all imported identifiers had been fully qualified during another phase of this thesis project, this would provide an accurate list of imported identifiers. Although the tool used to do the search did produce a complete list of all the identifiers, the list contained extraneous information. By using the Unix stream editor and some manual intervention, we were able to pare the list down so that it contained only the imported package names and identifiers. Appendix B contains a complete list of the STARS identifiers that are imported by the Saber user interface and the Hex bindings.

Once we had a complete list of all the imported STARS identifiers, the next task was to develop a basis for comparing the STARS identifiers with the SERC identifiers. For each identifier in Appendix P, we took the following steps:

1. *Identify the STARS source code statements that declared the imported identifier.* In many cases, we had to take this one step further. If the declaration involved a data type and the data type was an application defined data type, we traced the data type to its origin. Since application defined data types are ultimately based on a predefined Ada data type, this would provide a common basis for comparing the STARS and SERC identifiers.
2. *Search the SERC source code for an identifier with the same or similar name as the STARS identifier.* If one was found, we proceeded directly to the next step. Otherwise, we reviewed the function (semantics) of the STARS identifier and attempted to find a SERC identifier that performed a similar function. If one was found, we proceeded to the next step. Otherwise, the search for an equivalent identifier stopped here. Fortunately, the SERC bindings used naming conventions similar to the STARS bindings in most cases.
3. *Identify the SERC source code statements that declared the STARS identifier.* Like the first step, if the SERC declaration involved an application defined data type, we traced the data type to its origin.

At the end of each iteration of the previous task, we objectively compared the STARS and SERC identifiers for equivalence. We also subjectively evaluated the level of effort required to convert Saber from using an instance of the STARS identifier to using the SERC identifier. We created a conversion guide to document the basis on which the comparisons and evaluations were made. The conversion guide contains an alphabetical list of each STARS module and identifier referenced by Saber and the hex bindings. Figure 21 depicts one of the simpler entries from the conversion guide. First, each entry includes the STARS module and identifier name and the equivalent SERC module and identifier name, if any. In this particular case, "XT" is an abbreviation for the STARS module `X_Toolkit_Intrinsics.OSF`. Next, each entry includes a trace of the declarations on which the identifiers are defined including the name of the file(s) containing the declarations. Then, the apparent conversion steps are listed. The minimum conversion steps are (a) change the context clause from referencing the STARS module to reference the SERC

```

**EQUIVALENCE**
XT.Pixel == X_Lib.Pixel

**STARS DECLARATIONS**
boeing_xt.a:  subtype    PIXEL      is AFS_LARGE_NATURAL;
boeing_afs.a:  subtype AFS_LARGE_NATURAL is AFS_LARGE_INTEGER range
    0 .. AFS_LARGE_INTEGER'LAST;
boeing_afs.a:  type AFS_LARGE_INTEGER is new INTEGER;          -- 32 bits

**SERC DECLARATIONS**
sercx11/xlib/x_lib_.a:  type Pixel is new Int;
sercx11/xlib/x_configdep_.a: subtype Int is Long;--For Sun3, Sun4c, IBM, HP.
sercx11/xlib/x_configdep_.a: subtype Long is Int32;
sercx11/xlib/x_configdep_.a: type Int32 is range -2 ** 31 .. (2 ** 31) - 1;

**CONVERSION STEPS**
Convert statements involving different range of types.

**PREDICTED CONVERSION EFFORT**
Easy

```

Figure 21. Example Entry from the STARS to SERC Conversion Guide

module and (b) replace the fully qualified STARS identifier with the fully qualified SERC identifier. Additional conversion steps are noted by exception. Finally, each entry includes the predicted conversion effort, a relative measure of difficulty, which is explained below.

Table 2 summarizes the results of our evaluations. It shows the number of identifiers imported from the STARS modules and the predicted level of effort for converting from the STARS bindings to the SERC bindings. Generally speaking, the predicted the level of effort was based on two factors: equivalence of functionality and equivalence of data types. The specific criteria used for each level is as follows:

- *Easy*. The functionality of the identifiers is the same. The data types are the same, or they are scalar (integer, real, or enumeration) types.
- *Moderately difficult*. The functionality of the identifiers is the same. The identifiers are subprograms with a different number and range types, but the types can be converted readily.

Table 2. Conversion Level of Effort

STARS Modules	Levels of Effort				Total
	E	M	D	Q	
AFS_Basic_Types	5				5
Xlib	4			4	8
XM_Widget_Set	112	3		29	144
X_Toolkit_Intrinsics_OSF	21		2	18	41
X_Windows	22	3		6	31

- *Difficult.* The functionality of the identifiers is not the same, or the data types are so different that any statement that references the identifier will have to be modified.
- *Questionable.* An equivalent identifier has not been found, or the identifier relies upon an identifier whose equivalent identifier has not been found.

6.4.2 *Incremental Replacement of the STARS Identifiers.* Although the results from the previous phase were somewhat less than promising, we did perform some of the easy conversions, and attempted to perform one of the questionable conversions. As it turned out, however, even the easy conversions proved not to be easy. This was not because of any problem with the identifiers themselves. It was due to the relationship between the Saber user interface and the Hex bindings with the Boeing bindings. Since, the Hex bindings only used a few of the Boeing bindings, we decided to use Hex bindings as a test case. The plan was to iteratively convert the identifiers in the Hex bindings and conduct regression testing with each iteration. Unfortunately, we encountered actual/formal argument conflicts because the Saber user interface used those same identifiers. A temporary work-around was to go directly into the Boeing bindings and convert its identifiers. This proved successful for the easy identifiers, such as AFS.AFS_Large_Natural and Xlib.Null_Pixmap_ID. Unfortunately, when we reached the point that we needed to make one of the difficult conversions, XT.Arg_List, we were unable to complete the conversion. Figure 22 shows that XT.Arg_List and its apparent counterpart are composite data types with substantially different decompositions. To convert XT.Arg_List, not only would we have to convert the types on which it is based, but also we would have to modify every state-

```

**EQUIVALENCE**
XT.Arg_List == Xt.Xt_Ancillary_Types.Arg_List_Ptr (types differ)

**STARS DECLARATIONS**
boeing_xt.a: type ARG_LIST is access ARG_LIST_REC;
boeing_xt.a: type ARG_LIST_REC (ARG_COUNT : AFS_MEDIUM_NATURAL) is record
    LIST      : ARG_ARRAY (1..ARG_COUNT);
    INDEX     : AFS_MEDIUM_NATURAL := 0;
end record;
boeing_afs.a: subtype AFS_MEDIUM_NATURAL is AFS_MEDIUM_INTEGER range
    0 .. AFS_MEDIUM_INTEGER'LAST;
boeing_afs.a: type AFS_MEDIUM_INTEGER is new SHORT_INTEGER;      -- 16 bits
boeing_afs.a: type ARG_ARRAY is array(AFS_MEDIUM_POSITIVE range <>) of ARG_REC;
boeing_afs.a: subtype AFS_MEDIUM_POSITIVE is AFS_MEDIUM_INTEGER range
    1 .. AFS_MEDIUM_INTEGER'LAST;
boeing_afs.a: type AFS_MEDIUM_INTEGER is new SHORT_INTEGER;      -- 16 bits
boeing_afs.a: type ARG_REC is record
    NAME      : XTN_RESOURCE_STRING := NULL_ADDRESS;
    VALUE     : AFS_LARGE_INTEGER  := 0;
end record;
boeing_xt.a: subtype XTN_RESOURCE_STRING is SYSTEM.ADDRESS;
vads6/standard/system.a: type ADDRESS is private;
vads6/standard/system.a: type ADDRESS is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
vads6/standard/unsigned.a:-- type unsigned_integer is range 0 .. (2**32 - 1);
See XT.Null_Address
boeing_afs.a: type AFS_LARGE_INTEGER is new INTEGER;              -- 32 bits

**SERC DECLARATIONS**
sercx11/xt/xt_.a: subtype Arg_List_Ptr is Xt_Arg_List_Ptr;
sercx11/xt/xt_.a: type Xt_Arg_List_Ptr is access Xt_Arg_List;
sercx11/xt/xt_.a: subtype Arg_List is Xt_Arg_List;
sercx11/xt/xt_.a: type Xt_Arg_List is array (NATURAL range <>) of Xt_Arg;
sercx11/xt/xt_.a: type Xt_Arg is record
    Name      : Xt_String;
    Value     : Xt_Arg_Val;
end record;
sercx11/xt/xt_.a: subtype Xt_String is X_Lib.String_Pointer;
sercx11/xlib/x_lib_.a: type String_Pointer is access STRING;
sercx11/xt/xt_.a: subtype Xt_Arg_Val is X_Long_Integer;
sercx11/xlib/x_lib_.a: type X_Long_Integer is new Long;
sercx11/xlib/x_configdep_.a: subtype Long is Int32;
sercx11/xlib/x_configdep_.a: type Int32 is range -2 ** 31 .. (2 ** 31) - 1;

**CONVERSION STEPS**
Convert statements involving different types.

**PREDICTED CONVERSION EFFORT**
Difficult

```

Figure 22. XT.Arg_List Entry from the STARS to SERC Conversion Guide

ment that references a variable of type `XT.Arg_List`. Since other priorities demanded our attention, this incremental replacement task was set aside for the time being.

6.5 Summary

This chapter described the integrated implementation of the processors and presented an investigation of the feasibility of replacing the current Ada to X Windows interface. Since the pre-processors were designed with integration in mind, the actual act of integration proved to be fairly straightforward. The pre-processor was then implemented consistent with the principles of user interface design. To investigate the feasibility of replacing the current Ada to X Windows interface, we first compared identifiers from the STARS bindings with those from the SERC bindings, and then we incrementally replaced some STARS identifiers with SERC identifiers. Although we predicted it would be relatively easy to convert the majority of the identifiers, we also predicted that many of the conversions would be difficult or questionable at best. This proved to be the case when we did convert some of the easy identifiers and attempted to convert one of the difficult ones. The next chapter summarizes the thesis effort and presents conclusions and recommendations.

VII. Conclusions and Recommendations

7.1 Conclusions

This thesis effort completed the design and partially implemented the Saber user interface's pre-processor. Furthermore, this thesis effort integrated the designs and implementations of the pre-processor and the post-processor. These two steps have provided the means for the user to submit instructions to the game's assets and eliminated any inconsistency or redundancy between the two processors. The approach to achieve this goal included the following steps:

- Analyzed the problem domain/literature review. This step included a thorough review of Saber's development history and research into three of the integral components of this project: user interface design, the X Window System, and Ada to X Windows interfaces. The research into user interface design led to the design and implementation of a user interface that incorporates the goals and principles of "good" user interface design. Commensurate with the goals of user interface design, the X Window System proved to be an effective tool for constructing user-friendly interfaces. Finally, the Ada to X Windows interfaces offered the opportunity to take advantage of the the benefits afforded by the X Window System.
- Identified and consolidated the requirements. This step was based on an in-depth review of the previous thesis projects and consultation with the Saber development team. This effort resulted in a complete and current requirements specification of the four areas of the pre-processor that require mission input.
- Designed and integrated the requirements within the existing high-level object-oriented design. This effort employed a combination of reverse engineering and forward engineering to produce the integrated object-oriented design of the pre-processor and post-processor. Since part of the high-level design already existed, changes to it were kept to a minimum.
- Integrated the implementation of the processors. Since the pre-processors were designed with integration in mind, the actual act of integration proved to be fairly

straightforward. Two of the four mission input areas of the pre-processor were then implemented consistent with the principles of user interface design. The two mission input areas implemented were aircraft beddown missions and supply missions.

This thesis effort also investigated the feasibility of replacing the current Ada to X Windows interface, which uses Ada bindings to X Windows. This had the potential of enhancing the maintainability and functionality of the Saber user interface. To investigate the feasibility of replacing the current Ada to X Windows interface, we first compared identifiers from the STARS bindings with those from the SERC bindings, and then we incrementally replaced some STARS identifiers with SERC identifiers. Although we predicted it would be relatively easy to convert the majority of the identifiers, we also predicted that many of the conversions would be difficult or questionable at best. This proved to be the case when we did convert some of the easy identifiers and attempted to convert one of the difficult ones. Unfortunately, since other priorities demanded our attention, this incremental replacement task was set aside. Consequently, the potential benefits of replacing the current Ada to X Windows interface are still just that, potential benefits.

7.2 Recommendations

To achieve a fully functional Saber user interface, recommend the following additions be made:

- Implement the combat mission input areas of the pre-processor: aircraft missions and land unit missions.
- Implement all of the required status reports in on-screen or hard-copy form. Textual reports are essential; graphical charts and graphs should be considered.
- Implement an intelligence filtering mechanism to realistically restrict the information available to each side about the status and location of the opponent. A team should receive information about the opponent based on the intelligence gathering capabilities of its force.
- Minimize the resource consumption by the graphical user interface. Reducing the main memory requirement will enhance portability of the model and reducing the

central processing unit utilization will accelerate the model's response and turn-around times.

7.3 Summary

This thesis documents the integrated design and implementation of the Saber user interface which includes the pre-processor and post-processor. Although the user interface is not quite complete, this thesis effort has resulted in an user-friendly interface that is also understandable and maintainable.

Appendix A. *Saber Object Class Descriptions*

This appendix describes the application and Motif object classes which form the Saber user interface. The object classes are designed to be programming language independent. The attributes and methods are described for each class.

A.1 Application Classes

A.1.1 Airbase Class. Objects of this class contain information about the airbases covering the hexboard. This class has methods for creating, displaying, and erasing airbase symbols; and for displaying and erasing airbase status boards.

A.1.1.1 Attribute Types

Airbase_Pointer: This is a pointer to a list of aircraft bases. The following information is kept for each aircraft base:

- **Static Information:** base id, name, command, country, force, higher headquarters, total POL storage, maximum ramp space
- **Status Information:** ramp space available, alternate fields, intel index, number of enemy mines, status, MOPP posture, POL on base, POL in hard storage, maintenance personnel on hand, maintenance hours accumulated, amount of maintenance equipment, amount of spare parts, number of shelters, number of EOD crews, number of rapid runway repair (RRR) crews, an indicator of the base's visibility to the enemy, and the weather
- **Widget Information:** widget id of the form, symbol, and label widgets in addition to the widget id for the status window.

A.1.1.2 Methods. This section lists the methods for the Airbase class. The purpose, input parameters, and output parameters are given for each method.

Erase_Airbase_Status

- **Purpose:** This procedure closes a status window for a particular aircraft base.
- **Inputs:** Main Hexboard, Airbase Pointer for a particular unit
- **Outputs:** the Aircraft Base Pointer with the widget id of the status window removed

Erase_All_Airbases

- **Purpose:** This procedure erases all the aircraft bases (for a particular side) from the map.
- **Inputs:** Main Hexboard, Theater Map, Airbase List for a particular side (force)

- Outputs: none

Erase_Single_Airbase

- Purpose: This procedure erases a single aircraft base from the map. It also removes the base from the list of aircraft bases.
- Inputs: Main Hexboard, Theater Map, Airbase Pointer for a particular unit
- Outputs: none

Get_Airbase

- Purpose: This function returns a pointer to a specific aircraft base's information.
- Inputs: Airbase List, Base Id
- Outputs: pointer to the aircraft base with the specified Base Id

Initialize_Airbase_Symbols

- Purpose: This procedure creates the pixmaps for the Red and Blue airbase symbols.
- Inputs: Display Id, Red foreground and background colors, Blue foreground and background colors.
- Outputs: none

Is_Displaying_Status

- Purpose: This function returns an indication of whether a particular airbase has a status window open.
- Inputs: Airbase Pointer for a particular unit
- Outputs: "true" if the aircraft base has an open status window, "false" otherwise

Read_Airbase_Data

- Purpose: This procedure creates instances of type Airbase_Pointer by reading the specified database flat files.
- Inputs: Filenames for the Airbase, Depot, Runways, Alternate Bases, Airbase Aircraft, Airbase Weapons, and Weather database relations
- Outputs: Two variables of type Airbase_Pointer that point to a list of airbases. One pointer is returned for the Red bases and one for the Blue bases.

Show_Airbase_Status

- Purpose: This procedure opens a status window for a particular aircraft base.
- Inputs: Main Hexboard, Airbase Pointer for a particular unit
- Outputs: the Aircraft Base Pointer with the widget id of the status window added

Show_All_Airbases

- Purpose: This procedure displays all the aircraft bases (for a particular side) on the map.
- Inputs: Main Hexboard, Theater Map, Airbase List for a particular side (force)
- Outputs: none

Update_Airbase_Status

- Purpose: This procedure updates the status for a particular aircraft base. If the unit is currently displaying its status, the status window is also updated.
- Inputs: Main Hexboard, Airbase Pointer for a particular unit, new Status Information
- Outputs: the Aircraft Base Pointer with updated Status Information

A.1.2 Aircraft Mission Class. Objects of this class contain information about the aircraft missions covering the hexboard. This class has methods for creating, displaying, and erasing aircraft mission symbols; and for displaying and erasing aircraft mission status boards.

A.1.2.1 Attribute Types

Aircraft_Mission_Pointer: This is a pointer to a list of aircraft missions. The following information is kept for each aircraft mission:

- Static Information: aircraft package number, mission, target, requested time on target, altitude, hex level
- Aircraft Information: hex location, the starting number, current number, and types of aircraft flying as Primary, Suppression of Enemy Air Defense (SEAD), Electronic Counter Measures (ECM), Refueling, and Escort aircraft.
- Widget Information: widget id of the form, symbol, and label widgets in addition to the widget id for the status window.

A.1.2.2 Methods. This section lists the methods for the Aircraft Mission class. The purpose, input parameters, and output parameters are given for each method.

Add_Aircraft_Mission

- Purpose: This procedure adds an instance of type Aircraft_Mission_Pointer to the Aircraft Mission List.
- Inputs: Static Information, Aircraft Information, Aircraft Mission List, Main Hexboard, Theater Map, an indication to draw the mission on the game boards.
- Outputs: Aircraft Mission List with the new mission added

Erase_All_Aircraft_Missions

- Purpose: This procedure erases all the aircraft missions (for a particular side) from the map.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission List for a particular side (force)
- Outputs: none

Erase_Mission_Status

- Purpose: This procedure closes a status window for a particular aircraft mission.
- Inputs: Main Hexboard, Aircraft Mission Pointer for a particular unit
- Outputs: the Aircraft Mission Pointer with the widget id of the status window removed

Erase_Single_Aircraft_Mission

- Purpose: This procedure erases a single aircraft mission from the map. It also removes the mission from the list of aircraft missions.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission Pointer for a particular unit
- Outputs: none

Get_Aircraft_Mission

- Purpose: This function returns a pointer to a specific aircraft mission's information.
- Inputs: Aircraft Mission List, Mission Id
- Outputs: pointer to the aircraft mission with the specified Mission Id

Initialize_Aircraft_Symbols

- Purpose: This procedure creates the pixmaps for the Red and Blue aircraft mission symbols.
- Inputs: Display Id
- Outputs: none

Is_Displaying_Status

- Purpose: This function returns an indication of whether a particular unit has a status window open.
- Inputs: Aircraft Mission Pointer for a particular unit
- Outputs: "true" if the aircraft mission has an open status window, "false" otherwise

Move_Aircraft_Mission

- Purpose: This procedure moves a single aircraft mission on the map.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission Pointer for a particular unit, an indication to draw the mission on the game boards.
- Outputs: the Aircraft Mission Pointer with new location information

Set_GC_Aircraft_Symbols

- Purpose: This procedure sets the graphics context for the aircraft mission symbols.
- Inputs: X Windows Display, X Windows Drawable, Red Foreground and Background Colors, Blue Foreground and Background Colors
- Outputs: none

Show_All_Aircraft_Missions

- Purpose: This procedure displays all the aircraft missions (for a particular side) on the map.
- Inputs: Main Hexboard, Theater Map, Aircraft Mission List for a particular side (force)
- Outputs: none

Show_Mission_Status

- Purpose: This procedure opens a status window for a particular aircraft mission.
- Inputs: Main Hexboard, Aircraft Mission Pointer for a particular unit
- Outputs: the Aircraft Mission Pointer with the widget id of the status window added

Update_Mission_Status

- Purpose: This procedure updates the status for a particular aircraft mission. If the unit is currently displaying its status, the status window is also updated.
- Inputs: Main Hexboard, Aircraft Mission Pointer for a particular unit, new Status Information
- Outputs: the Aircraft Mission Pointer with updated Status Information

A.1.3 Animation Controller Class. This class instantiates objects throughout the previous simulation day.

A.1.3.1 Attribute Types

Event_Type: This type enumerates the kinds of events that the animation task must act upon.

History_String_Type: This type contains a line of text from the mission history file.

Saber_Animation: This type is an anonymous task type that contains entry points for controlling the instantiation of animation objects.

A.1.3.2 Methods. This section lists the methods for the Animation Controller class. The purpose, input parameters, and output parameters are given for each method.

Is_Active

- **Purpose:** This function returns "true" if the player wants to continue the animation. Otherwise, it returns "false".
- **Inputs:** none
- **Outputs:** boolean value indicating whether the player wants to continue the animation.

Saber_Animation.Fill_In_Theater_Map

- **Purpose:** This task entry point calls a procedure to display the theater map hexes.
- **Inputs:** Top level widget, hexboard, and event
- **Outputs:** Theater map

Saber_Animation.Pause_Animation

- **Purpose:** This task entry point sets the value that indicates a player's desire to pause the animation to "true".
- **Inputs:** none
- **Outputs:** none

Saber_Animation.Redisplay_Terrain

- **Purpose:** This task entry point calls a procedure to redisplay the terrain objects.
- **Inputs:** Game board
- **Outputs:** none

Saber_Animation.Redisplay_Units

- **Purpose:** This task entry point calls a procedure to redisplay the air and ground unit objects.

- Inputs: none
- Outputs: none

Saber_Animation.Resume_Animation

- Purpose: This task entry point sets the value that indicates a a players desire to pause the animation to "false".
- Inputs: none
- Outputs: none

Saber_Animation.Resume_Window_Operations

- Purpose: This task entry point resumes the momentarily suspended animation.
- Inputs: none
- Outputs: none

Saber_Animation.Show_Hex_Info

- Purpose: This task entry point calls procedures to retrieve the position of a hex and display information about the hex.
- Inputs: Top level widget, hexboard, and event
- Outputs: none

Saber_Animation.Start_Animation

- Purpose: This task entry point reads the database files and instantiates objects reflecting the status of the battlefield at the start of the animation period.
- Inputs: Top level widget and hex board
- Outputs: none

Saber_Animation.Stop_Animation

- Purpose: This task entry point calls procedures to erase all air and ground units from the display and to close the mission history file.
- Inputs: none
- Outputs: none

Saber_Animation.Suspend_Window_Operations

- Purpose: This task entry point momentarily suspends the animation.
- Inputs: none
- Outputs: none

Saber_Animation.Weather_Display_Toggle_Changed

- Purpose: This task entry point calls procedures to either erase or display the weather background.
- Inputs: Parent
- Outputs: none

A.1.4 Day End Controller Class. This class instantiates objects representing the state of the simulation at the end of the previous simulation day.

A.1.4.1 Attribute Types

none

A.1.4.2 Methods. This section lists the methods for the Day End Controller class. The purpose, input parameters, and output parameters are given for each method.

Clear_Main_Hex_Board

- Purpose: This procedure calls procedures to erase all air and ground unit objects from the display.
- Inputs: none
- Outputs: none

Initialize_Main_Hex_Board

- Purpose: This procedure reads the database files and instantiates objects reflecting the status of the battlefield at the end of the previous simulation day.
- Inputs: Top level widget and game board
- Outputs: none

Redisplay_Main_Hex_Board

- Purpose: This procedure calls procedures to redisplay all air unit, and ground unit, and terrain objects.
- Inputs: Top level widget and game board
- Outputs: none

Redisplay_Terrain

- Purpose: This procedure calls procedures to redisplay all terrain objects.
- Inputs: Top level widget and game board
- Outputs: none

Show_Hex_Info

- Purpose: This procedure calls procedures to retrieve the position of a hex and display information about the hex.
- Inputs: Top level widget, hexboard, and event
- Outputs: none

Show_Theater_Map_Data

- Purpose: This procedure calls a procedure display the theater map.
- Inputs: Theater map
- Outputs: none

Weather_Display_Toggle_Changed

- Purpose: This task entry point calls procedures to either erase or display the weather background.
- Inputs: Parent
- Outputs: none

A.1.5 Forces Class. Objects of this class contain information about the countries and the forces to which they belong. This class has methods for reading the force data file and retrieving the force identifier (side) of a given country.

A.1.5.1 Attribute Types

Force.Pointer: This is a pointer to a list of force records. The following information is kept for each country:

- Static Information: country and force.

A.1.5.2 Methods. This section lists the methods for the Forces class. The purpose, input parameters, and output parameters are given for each method.

Read_Forces_Data

- Purpose: This procedure creates instances of Force.Pointer by reading the specified database flat file.
- Inputs: Filename for the Forces database relation.
- Outputs: A global variable of type Forces.Pointer that points to a list of forces.

A.1.6 Help Class. Objects of this class contain information about air and ground units and the assets available at those units. This class has methods for creating the information needed for context sensitive help.

A.1.6.1 Attribute Types

Airbase_Pooter: This is a pointer to a list of airbases. The following information is kept for each airbase:

- Static Information: airbase id, full airbase name, abbreviated airbase name, country, location

Airbase_Aircraft_Pooter: This is a pointer to a list of airbases. The following information is kept for each aircraft type at each airbase:

- Static Information: airbase id, aircraft name, aircraft quantity

Aircraft_Type_Pooter: This is a pointer to a list of aircraft types. The following information is kept for each aircraft type:

- Static Information: aircraft name, force, full aircraft name

Clock: This is a record of database clock relation and contains day and time period of the database.

A.1.6.2 Methods. This section lists the methods for the Help class. The purpose, input parameters, and output parameters are given for each method.

A_String_To_Addr

- Purpose: This function converts a variable of type A_Strings.A_String to a variable of type System.Address.
- Inputs: A variable of type A_Strings.A_String
- Outputs: A variable of type System.Address

Addr_To_A_String

- Purpose: This function converts a variable of type System.Address to a variable of type A_Strings.A_String.
- Inputs: A variable of type System.Address
- Outputs: A variable of type A_Strings.A_String

Create_Airbase_Help

- Purpose: This procedure calls a procedure to create a Motif scrolled list, extracts the airbase data for a given side from a list of type Airbase_Pooter, and then calls a procedure to add the airbase information to the scrolled list.
- Inputs: Pointer to an airbase list, parent of the scrolled list, side, and callback address for the scrolled list
- Outputs: The scrolled list

Create_Aircraft_Type_Help

- Purpose: This procedure calls a procedure to create a Motif scrolled list, extracts the aircraft type data for a given airbase from lists of type Aircraft_Type_Pointer and Airbase_Aircraft_Pointer, and then calls a procedure to add the aircraft type information to the scrolled list.
- Inputs: Pointer to an aircraft type list, pointer to an airbase aircraft list, name of an airbase, parent of the scrolled list, and callback address for the scrolled list
- Outputs: The scrolled list

Create_Help_Button

- Purpose: This procedure creates a default menu "HELP" button.
- Inputs: Parent and message
- Outputs: none

Create_Help_Screen

- Purpose: This procedure creates an Information Dialog box to display a help message.
- Inputs: Parent and message
- Outputs: none

Read_Airbase_Relation

- Purpose: This procedure creates instances of type Airbase_Pointer by reading the specified database flat files.
- Inputs: Filename for the Airbase database relation
- Outputs: A variable of type Airbase_Pointer that point to a list of airbases

Read_Airbase_Aircraft_Relation

- Purpose: This procedure creates instances of type Airbase_Aircraft_Pointer by reading the specified database flat files.
- Inputs: Filename for the Airbase Aircraft database relation
- Outputs: A variable of type Airbase_Aircraft_Pointer that points to a list of airbases and the aircraft at each airbase.

Read_Aircraft_Relation

- Purpose: This procedure creates instances of type Aircraft_Type_Pointer by reading the specified database flat files.
- Inputs: Filename for the Aircraft Type database relation

- **Outputs:** A variable of type `Aircraft_Type_Pointer` that point to a list of aircraft types

Read_Clock_Relation

- **Purpose:** This procedure creates instances of type `Clock` by reading the specified database flat file
- **Inputs:** Filename for the `Clock` database relation
- **Outputs:** A variable of type `Clock`

Validate_Airbase

- **Purpose:** This procedure determines if a given airbase designator is valid by comparing the designator against a list of valid airbase names and ids. It then returns the valid airbase name and id, if any, and a boolean indicating whether the designator was valid.
- **Inputs:** Pointer to an airbase list and airbase designator
- **Outputs:** Airbase name, airbase identifier, and successful

Validate_Aircraft_Quantity

- **Purpose:** This procedure determines if the aircraft quantity requested is less than or equal to the aircraft quantity available. It then returns a valid aircraft quantity requested string, if any, and a boolean indicating whether the aircraft quantity requested was valid.
- **Inputs:** Aircraft quantity requested, aircraft quantity available, length of valid aircraft quantity requested string
- **Outputs:** Valid aircraft quantity requested string and successful

Validate_Aircraft_Type

- **Purpose:** This procedure determines if a given aircraft type designator is valid by comparing the designator against a list of valid aircraft for a particular base. It then returns the valid airbase type name, common aircraft type name, and quantity, if any, and a boolean indicating whether the designator was valid.
- **Inputs:** Pointer to an particular airbase's aircraft list and an aircraft type designator
- **Outputs:** Abbreviated aircraft type name, common aircraft type name, quantity of the aircraft type, and successful

A.1.7 Hexboard Class. Objects of this class are hexboards in which terrain and/or units can be displayed. The methods create and manipulate the visible portion of the hexboard.

A.1.7.1 Attribute Types

Game_Board_Type: This is the type for the Game Board (hexboard) object and is instantiated through a call to *Create_Hexboard*. Variables of this type contain the widget id, width, height of the hexboard as well as the current X, Y, W, H locations. Additionally, they contain the graphics context for the location box and an indication of whether or not the first exposure of the hexboard has occurred.

Hexboard_Client_Data_Type: This is the type for the client data to be passed to hexboard callback procedures. It consists of **Game_Board_Type** instantiations for the main hexboard and the theater map. It also contains the widget id of the scrolled window widget containing the main hexboard.

A.1.7.2 Methods. This section lists the methods for the Hexboard class. The purpose, input parameters, and output parameters are given for each method.

Create_Hexboard

- **Purpose:** This function creates a hexboard of the specified size and with the specified options.
- **Inputs:** Parent, Name, Background Color, Hex Outline Color, Stacking Dot Color, number of hexes in the X and Y directions, Hex Radius, and whether or not to display Hex Labels
- **Outputs:** an instantiated object of type **Game_Board_Type**

Draw_Location_Box

- **Purpose:** This procedure draws the location box in the theater map.
- **Inputs:** Hexboard Client Data
- **Outputs:** none

Set_GC_Location_Box

- **Purpose:** This procedure sets the graphics context for the location box. The location box is drawn in the theater map and shows the portion of the theater that is being displayed in the main hexboard
- **Inputs:** Theater Map, X Windows Display, X Windows Drawable, Location Box Color
- **Outputs:** none

Set_Theater_Map_Active

- **Purpose:** This procedure sets a flag to indicate that the theater map is being displayed.
- **Inputs:** none

- Outputs: none

Theater_Map_Button_Press

- Purpose: This procedure adjusts the visible portion of the main hexboard as a result of the user pressing the mouse button while the sprite is inside the theater map.
- Inputs: Hexboard Client Data
- Outputs: none

Theater_Map_Is_Active

- Purpose: This function returns “true” if the theater map is being displayed. Otherwise, it returns “false”.
- Inputs: none
- Outputs: boolean value indicating if theater map is currently being displayed

A.1.8 Game Player Class. This class contains information about the person playing the game. This class has methods for setting and retrieving this information.

A.1.8.1 Attribute Types

Current_Day_Type: integer representing the current day relative to the start of the battle

Player_Side_Type: indicates whether the player is on the RED, BLUE, or WHITE team

Seminar_Number_Type: string for the seminar number of the class

A.1.8.2 Methods. These are the methods for the Game Player class. The purpose, input parameters, and output parameters are given for each method.

Fetch_Start_Up_Form

- Purpose: generates the start up form to retrieve player information
- Inputs: addresses of “CONTINUE” and “QUIT” callback procedures
- Outputs: widget id of the start up form

Get_Current_Day

- Purpose: function that returns the current day
- Inputs: none
- Outputs: none

Get_Seminar_Number

- Purpose: function that returns the seminar number

- Inputs: none
- Outputs: none

Is_Blue

- Purpose: function that returns true if player side is BLUE
- Inputs: none
- Outputs: none

Is_Red

- Purpose: function that returns true if player side is RED
- Inputs: none
- Outputs: none

Is_White

- Purpose: function that returns true if player side is WHITE
- Inputs: none
- Outputs: none

Set_User_Values

- Purpose: procedure that extracts and saves the values for the Seminar Number, Current Day, and Player side from the widgets.
- Inputs: none
- Outputs: none

A.1.9 Ground Unit Class. Objects of this class contain information about the ground units covering the hexboard. The methods read and display the ground units and their status.

A.1.9.1 Attribute Types

Ground_Unit_Pointer: This is a pointer to a list of ground units. The following information is kept for each ground unit:

- Static Information: unit designator, unit type, country, force, corps id, supported units, supporting units
- Status Information: mission, target number, combat power, firepower, surface-to-air index, total ammunition, total hardware, total petroleum-oil-lubricants (POL), amount of water, amount of engineer support, intel index, visible to enemy indicator, weather, hex location
- Widget Information: widget id of the form, symbol, and label widgets in addition to the widget id for the status window.

A.1.9.2 Methods. This section lists the methods for the Ground Unit class. The purpose, input parameters, and output parameters are given for each method.

Erase_All_Ground_Units

- Purpose: This procedure erases all the ground units (for a particular side) from the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit List for a particular side (force)
- Outputs: none

Erase_Single_Ground_Unit

- Purpose: This procedure erases a single ground unit from the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit Pointer for a particular unit
- Outputs: none

Erase_Unit_Status

- Purpose: This procedure closes a status window for a particular ground unit.
- Inputs: Main Hexboard, Ground Unit Pointer for a particular unit
- Outputs: the Ground Unit Pointer with the widget id of the status window removed

Get_Ground_Unit

- Purpose: This function returns a pointer to a specific ground unit's information.
- Inputs: Ground Unit List, Unit Id
- Outputs: pointer to the ground unit with the specified Unit Id

Initialize_Ground_Symbols

- Purpose: This procedure creates the pixmaps for the Red and Blue ground unit symbols.
- Inputs: Display Id
- Outputs: none

Is_Displaying_Status

- Purpose: This function returns an indication of whether a particular unit has a status window open.
- Inputs: Ground Unit Pointer for a particular unit
- Outputs: "true" if the ground unit has an open status window, "false" otherwise

Move_Ground_Unit

- Purpose: This procedure moves a single ground unit on the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit Pointer for a particular unit
- Outputs: the Ground Unit Pointer with new location information

Read_Ground_Unit_Data

- Purpose: This procedure creates instances of type Ground_Unit_Pointer by reading the specified database flat files.
- Inputs: Filenames for the Land Unit, Unit Supports, Move, Move LNLT, Unit Components, and Unit G2A database relations.
- Outputs: Two variables of type Ground_Unit_Pointer that point to a list of ground units. One pointer is returned for the Red units and one for the Blue units.

Set_GC_Ground_Symbols

- Purpose: This procedure sets the graphics context for the ground unit symbols.
- Inputs: X Windows Display, X Windows Drawable, Red Foreground and Background Colors, Blue Foreground and Background Colors
- Outputs: none

Show_All_Ground_Units

- Purpose: This procedure displays all the ground units (for a particular side) on the map.
- Inputs: Main Hexboard, Theater Map, Ground Unit List for a particular side (force)
- Outputs: none

Show_Unit_Status

- Purpose: This procedure opens a status window for a particular ground unit.
- Inputs: Main Hexboard, Ground Unit Pointer for a particular unit
- Outputs: the Ground Unit Pointer with the widget id of the status window added

Update_Unit_Status

- Purpose: This procedure updates the status for a particular ground unit. If the unit is currently displaying its status, the status window is also updated.
- Inputs: Main Hexboard, Ground Unit Pointer for a particular unit, new Status Information
- Outputs: the Ground Unit Pointer with updated Status Information

A.1.10 Main Controller Class. This class instantiates objects that are common to the Animation Controller and the Day End Controller.

A.1.10.1 Attribute Types

Color_Resources_Record: This is the type containing entries for storing the color values for the various graphical objects to be displayed in the hex board.

A.1.10.2 Methods. This section lists the methods for the Main Controller class. The purpose, input parameters, and output parameters are given for each method.

Create_Main_Game_Board

- **Purpose:** This procedure creates and displays the main game board.
- **Inputs:** none
- **Outputs:** none

Create_Top_Level_Widget

- **Purpose:** This procedure calls `Xt_Initialize` to create the top level widget.
- **Inputs:** The number of arguments specified on the command line and the actual arguments specified on the command line.
- **Outputs:** none

Exit_Saber

- **Purpose:** This procedure is a callback from the user clicking on "EXIT" from the main menubar. This procedure stops the animation (if active), closes any open files, and raises the "XT_QUIT_X_REQUEST" exception to exit the `Xt_Main_Loop`.
- **Inputs:** none
- **Outputs:** none

Initialize_Color_Resources

- **Purpose:** This procedure calls `Get_Color_Resources` to read in values to override the default color values for the various graphical objects to be displayed in the hex board. The user specifies values for these color resources with case-sensitive entries in the `.Xdefaults` file located in the root directory.
- **Inputs:** none
- **Outputs:** none

Initialize_Graphics_Contexts

- Purpose: This procedure calls procedures to initialize the symbols to be displayed including ground unit, airbase, and aircraft package symbols.
- Inputs: none
- Outputs: none

Realize_Top_Level_Widget

- Purpose: This procedure calls Xt_Realize_Widget to make the Top_Level_Widget and all of its children appear on the screen. It also converts the Display_Id and Drawable_Id from the types used by Boeing's bindings to the types needed by SAIC's bindings to Xlib.
- Inputs: none
- Outputs: none

A.1.11 Main Procedure Class. This class displays the simulation startup form and instantiates the simulation startup task that processes X Windows events.

A.1.11.1 Attribute Types

none

A.1.11.2 Methods. This section lists the methods for the Main Procedure class. The purpose, input parameters, and output parameters are given for each method.

Saber_Main

- Purpose: This procedure displays the simulation startup form and instantiates the simulation startup task that processes X Windows events.
- Inputs: none
- Outputs: none

A.1.12 Mission Board Class. Objects of this class are mission input boards. This class has methods for creating, displaying, and erasing mission input boards.

A.1.12.1 Attribute Types

none

A.1.12.2 Methods. This section lists the methods for the Mission Board class. The purpose, input parameters, and output parameters are given for each method.

Create_Aircraft_Mission_Board

- Purpose: This function creates and displays the aircraft mission input board.

- Inputs: Parent and address of "CONFIRM" quit callback procedure.
- Outputs: none

Create_Beddown_Mission_Board

- Purpose: This procedure creates and displays the beddown mission input board.
- Inputs: Parent and address of "CONFIRM" quit callback procedure.
- Outputs: none

Create_Land_Unit_Mission_Board

- Purpose: This function creates and displays the land unit mission input board.
- Inputs: Parent and address of "CONFIRM" quit callback procedure.
- Outputs: none

Create_Supply_Mission_Board

- Purpose: This procedure creates and displays the supply mission input board.
- Inputs: Parent and address of "CONFIRM" quit callback procedure.
- Outputs: none

A.1.13 Report Class. Objects of this class contain status information about ground units, aircraft missions, and airbases. The methods display and print the various reports.

A.1.13.1 Attribute Types

Report_Pointer: This is a pointer to a list of report names.

A.1.13.2 Methods. This section lists the methods for the Report class. The purpose, input parameters, and output parameters are given for each method.

Display_Options_Menu

- Purpose: This procedure displays the main report options menu.
- Inputs: none
- Outputs: none

Print_Report

- Purpose: This function sends a list of reports to a printer.
- Inputs: Report List
- Outputs: none

Select_Standard_Set

- Purpose: This function allows the user to select a standard set of reports for daily printing.
- Inputs: none
- Outputs: Report List

View_Report

- Purpose: This procedure opens a window in which a report is displayed.
- Inputs: Report Name to display
- Outputs: none

A.1.14 Terrain Class. Objects of this class contain information about the terrain covering the hexboard. The methods read and display the terrain information.

A.1.14.1 Attribute Types

City_Pointer: This is a pointer to a list of cities. The following information is kept for each city: name, hex location, size, population, and whether or not it is a capital.

Hex_Array_Type: Variables of this type are arrays of asset information for the hexes. The asset information includes center hex id, sides of the hex that form part of air hex borders, force, country, weather zone, weather, intel index, combat power in, combat power out, terrain type, amount of forestation, pie trafficability for each hex side, hex sides containing roads, railroads, and pipelines.

Hex_Range: Variables of this type are integers between 0 and 99.

Neighbor_Array_Type: Variables of this type are arrays, indexed by neighbor id, that contain the two hexes and their hex sides that are neighbors.

Neighbor_List_Pointer: Variables of this type are pointers to a linked list of neighbor id's. This type is used for FEBA and country border lists.

Neighbor_Range: Variables of this type are integers between 1 and 9999.

Obstacle_Pointer: This is a pointer to a list of hexside obstacles. The following information is kept for each obstacle: obstacle id, neighbor id, name, and visibility to BLUE and RED forces.

River_Segment_Pointer: This is a pointer to a list of neighbor id's and river sizes that form the rivers.

Terrain_Object_Type: This is the type for the Terrain object and is instantiated through a call to *Read_Terrain_Data*. Variables of this type contain all terrain information about a hexboard. This type consists of a consolidation (i.e., record or structure) of the following variables listed with their corresponding types in parenthesis:

Hex_Array (Hex_Array_Type)	Border_List (Neighbor_List_Pointer)
Neighbor_Array (Neighbor_Array_Type)	FEBA_List (Neighbor_List_Pointer)
Obstacle_List (Obstacle_List_Pointer)	City_List (City_Pointer)
River_Segment_List (River_Segment_Pointer)	Max_X (Hex_Range)
Max_Neighbor (Neighbor_Range)	Max_Y (Hex_Range)

A.1.14.2 Methods. This section lists the methods for the Terrain class. The purpose, input parameters, and output parameters are given for each method.

Erase_Bridge

- Purpose: This procedure removes a single bridge from the given hexboard and the Terrain data structure.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y coordinates for each hex, Hex Sides
- Outputs: none

Erase_Minefield

- Purpose: This procedure removes a single minefield from the given hexboard and the Terrain data structure.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y coordinates, Hex Side
- Outputs: none

Erase_Weather_Data

- Purpose: This procedure removes the current weather from the given hexboard.
- Inputs: Hexboard Widget, Terrain Object
- Outputs: none

Flash_Hex_Background

- Purpose: This procedure rapidly changes the background color of a user specified hexagon to indicate the hex or units in the hex have been attacked.
- Inputs: Hex X and Y coordinates
- Outputs: none

Read_Terrain_Data

- Purpose: This procedure creates instances of variables of type Terrain_Object_Type by reading the specified database flat files.

- Inputs: Filenames for the Hex, Travel, City, Hexside Assets, FEBA, Roads, Railroads, and Pipelines database relations.
- Outputs: A variable of type `Terrain_Object_Type` instantiated with data from the input files.

Show_Bridge

- Purpose: This procedure adds a bridge to specific sides of two hexes on the given hexboard. The hex is redrawn if the user is currently displaying bridges.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y coordinates for each hex, Hex Sides, Terrain Display Toggle Button List
- Outputs: none

Show_Minefield

- Purpose: This procedure adds a minefield to a specific side of a hex on the given hexboard. The hex is redrawn if the user is currently displaying minefields.
- Inputs: Hexboard Widget, Terrain Object, Hex X and Y Coordinates, Hex Side, Terrain Display Toggle Button List
- Outputs: none

Show_Terrain_Data

- Purpose: This procedure displays all of the terrain data (except the weather) on the given hexboard subject to the current terrain display toggle buttons.
- Inputs: Hexboard Widget, Terrain Object, Terrain Display Toggle Button List
- Outputs: none

Show_Weather_Data

- Purpose: This procedure displays the current weather on the given hexboard.
- Inputs: Hexboard Widget, Terrain Object
- Outputs: none

A.2 Motif Classes

A.2.1 List Widget Class. Objects of this class are Motif scrolled lists. This class has methods for creating scrolled lists; adding, deleting, and replacing items in the scrolled lists.

A.2.1.1 Attribute Types

none

A.2.1.2 Methods. This section lists the methods for the List Widget class. The purpose, input parameters, and output parameters are given for each method.

Add_Scrolled_List_Item

- Purpose: This procedure adds an item to a scrolled list.
- Inputs: Item and visible item count
- Outputs: Scrolled list

Create_Scrolled_List

- Purpose: This function creates a scrolled list.
- Inputs: Parent, name, visible item count, width and address of a callback procedure.
- Outputs: Scrolled list

Delete_All_Scrolled_List_Items

- Purpose: This procedure deletes all the items from a scrolled list.
- Inputs: Scrolled list
- Outputs: none

Get_Scrolled_List_Item_Count

- Purpose: This function returns the number of items in a scrolled list.
- Inputs: Scrolled list
- Outputs: Scrolled list item count

Get_Scrolled_List_Selected_Item_Count

- Purpose: This function returns the number of selected (highlighted) items in a scrolled list.
- Inputs: Scrolled list
- Outputs: Scrolled list selected item count

Replace_Scrolled_List_Item

- Purpose: This procedure replaces an item in a scrolled list.
- Inputs: Scrolled list, replacement item, replacement position
- Outputs: none

A.2.2 Manager Widget Class. Objects of this class are instantiations of subclasses of the Motif manager widget class. This class has methods for instantiating subclasses of the manager widget class including dialogs, labels, and pushbuttons.

A.2.2.1 Attribute Types

none

A.2.2.2 Methods. This section lists the methods for the Manager Widget class. The purpose, input parameters, and output parameters are given for each method.

Add_Label_To_Row_Column

- Purpose: This function adds a text label to a row column widget.
- Inputs: Parent, name, text, x position, y position
- Outputs: label

Attach_Widget_To_Top_Widget

- Purpose: This procedure attaches a widget to the widget above it.
- Inputs: Top widget and bottom widget
- Outputs: none

Create_Bulletin_Dialog

- Purpose: This function creates a bulletin board dialog.
- Inputs: Parent and name
- Outputs: Bulletin board dialog

Create_Button_Column

- Purpose: This function creates a row column widget designed for the purpose of containing a column of pushbuttons.
- Inputs: Parent, name, top widget, top offset, left widget, and left offset
- Outputs: Row column widget

Create_Error_Dialog

- Purpose: This function creates an error dialog.
- Inputs: Parent, error message, and address of a callback procedure
- Outputs: Error dialog

Create_Form

- Purpose: This function creates a form.
- Inputs: Parent and name
- Outputs: Form

Create_Form_Title_Row_Column

- Purpose: This function creates a row column widget designed for the purpose of containing a form title.
- Inputs: Parent and name
- Outputs: Row column widget

Create_Frame

- Purpose: This function creates a frame widget.
- Inputs: Parent, name, shadow thickness
- Outputs: Frame

Create_Information_Dialog

- Purpose: This function creates an information dialog.
- Inputs: Parent, information message, and address of a callback procedure
- Outputs: Information dialog

Create_Push_Button

- Purpose: This function creates a push button widget.
- Inputs: Parent, name, address of a callback procedure, address of the client data for the callback procedure, x position, y position, height, width
- Outputs: Push button

Create_Row_Column

- Purpose: This function creates a row column widget.
- Inputs: Parent, name, top widget, and top offset
- Outputs: Row column widget

Create_Text

- Purpose: This function creates a text widget.
- Inputs: Parent, name, default text, top offset, x position, y position, width, and address of a callback procedure
- Outputs: Text widget

Create_Warning_Dialog

- Purpose: This function creates an warning dialog.
- Inputs: Parent, warning message, and addresses of two callback procedures
- Outputs: Warning dialog

Replace_Text

- Purpose: This procedure replaces the text in a text widget.
- Inputs: Parent and default text
- Outputs: none

Set_Widget_Position

- Purpose: This procedure sets the position of a widget with respect to its parent.
- Inputs: Widget, x position, y position
- Outputs: Widget

Set_Widget_Traversal

- Purpose: This procedure specifies whether a widget should be traversed by the display pointer.
- Inputs: Widget and boolean
- Outputs: Widget

A.2.3 Menubar Class. Objects of this class are menubar widgets that contain various pulldown menus. The methods create the menubar and allow for the addition of pulldown menus and the items on the pulldown menus.

A.2.3.1 Attribute Types

none

A.2.3.2 Methods. This section lists the methods for the Menubar class. The purpose, input parameters, and output parameters are given for each method.

Add_Pulldown_Menu_Item

- Purpose: This function adds a menu item to a pulldown menu. A pushbutton is created for the menu item with the specified callback.
- Inputs: Pulldown Menu Widget, Item Name, Callback Address
- Outputs: widget id of the newly created pushbutton

Create_Menubar

- Purpose: This function creates a menubar with a help pulldown menu on the far right side. Other pulldown menus can be added through the *Create_Pulldown_Menu* procedure.
- Inputs: Parent Widget, Help Message
- Outputs: widget id of menubar

Create_Pulldown_Menu

- Purpose: This function creates a cascade button on the specified menubar and a pulldown menu to hang off of it.
- Inputs: Parent Menubar Widget, Name On Menubar, Mnemonic, Pulldown Title
- Outputs: widget id of the pulldown menu

A.2.4 Toggle Button Board Class. Objects of this class are bulletin board widgets that contain various toggle buttons for various custom settings. The methods create and manipulate the toggle button boards. print the various reports.

A.2.4.1 Attribute Types

Button_List: This is a pointer to a list of button records.

Button_Record: This is a collection of information about a toggle button. It includes such information as: the button name, widget id, current value, new value, and a value changed callback address.

A.2.4.2 Methods. This section lists the methods for the Toggle Button Board class. The purpose, input parameters, and output parameters are given for each method.

Clear_Button_List

- Purpose: This procedure empties a Button List so it can be reused.
- Inputs: Button List
- Outputs: Button List

Create_Toggle_Button_Board

- Purpose: This procedure creates a bulletin board for a set of toggle buttons.
- Inputs: Button List, Board Title, Instructions, OK Callback Address, CANCEL Callback Address, HELP Callback address
- Outputs: none

Make_Button_List

- Purpose: This procedure creates an empty Button List.
- Inputs: none
- Outputs: pointer to an empty Button List

Reset_Toggle_Values

- Purpose: This procedure resets the values for the toggle buttons back to the state they were in before the toggle button bulletin board was created.

- Inputs: Button List
- Outputs: Button List with values reset to their initial state

Set_Button

- Purpose: This procedure adds information for a new button to the specified Button List
- Inputs: Button List, Button Record information
- Outputs: Button List with new Button Record added

Set_New_Toggle_Values

- Purpose: This procedure sets the new values for the toggle buttons to the state they were in when the user closed the toggle button bulletin board.
- Inputs: Button List
- Outputs: Button List with values set to their new state.

Toggle_Button_Value_Changed

- Purpose: This procedure handles *XmNvalueChanged* callbacks. It records the new value of a toggle button.
- Inputs: Button List
- Outputs: Button List with new value

Appendix B. *STARS Identifiers Imported by the Saber User Interface*

This appendix lists the STARS identifiers that are imported by the Saber user interface and the Hex bindings. The identifiers are grouped by the source library, Boeing or SAIC, and by the module (package) that declared them.

B.1 Boeing Binding Library Packages

- **AFS_Basic_Types Identifiers**
 - AFS_C_Unsigned_Short
 - AFS_Large_Integer
 - AFS_Large_Natural
 - AFS_Medium_Natural
 - AFS_Medium_Positive
- **Xlib Identifiers**
 - Default_Depth
 - Display_Pointer
 - Drawable
 - Null_Pixmap_Id
 - Pixmap_Id
 - Root_Window
 - Window_Id
 - X_Sync
- **XM_Widget_Set Identifiers**
 - XmN_activate_Callback
 - XmN_alignment
 - XmN_allow_Shell_Resize
 - XmN_background
 - XmN_bottom_Attachment
 - XmN_bottom_Widget
 - XmN_Cancel_Callback
 - XmN_Columns
 - XmN_Command_Changed_Callback
 - XmN_Default_Action_Callback
 - XmN_dialog_Style
 - XmN_Double_Click_Interval
 - XmN Editable
 - XmN_Edit_Mode
 - XmN_height
 - XmN_Horizontal_Scroll_Bar
 - XmN_indicator_On
 - XmN_indicator_Type
 - XmN_Item_Count
 - XmN_label_Pixmap
 - XmN_label_String
 - XmN_label_Type
 - XmN_left_Attachment
 - XmN_Left_Offset
 - XmN_left_Widget
 - XmN_List_Size_Policy
 - XmN_Map_Callback
 - XmN_margin_Height
 - XmN_margin_Left
 - XmN_margin_Right
 - XmN_margin_Width
 - XmN_maximum
 - XmN_max_Length
 - XmN_menu_Help_Widget

- XmN_Message_String
- XmN_mnemonic
- XmN_no_Resize
- XmN_OK_Callback
- XmN_OK_Label_String
- XmN_orientation
- XmN_Packing
- XmN_processing_Direction
- XmN_radio_Always_One
- XmN_radio_Behavior
- XmN_right_Attachment
- XmN_right_Widget
- XmN_scrolling_Policy
- XmN_Scroll_Bar_Display_Policy
- XmN_scroll_Horizontal
- XmN_scroll_Vertical
- XmN_Selected_Item_Count
- XmN_Selection_Policy
- XmN_sensitive
- XmN_separator_Type
- XmN_set
- XmN_Shadow_Thickness
- XmN_show_Value
- XmN_Single_Selection_Callback
- XmN_spacing
- XmN_sub_Menu_Id
- XmN_Text_Columns
- XmN_title
- XmN_top_Attachment
- XmN_top_Offset
- XmN_top_Widget
- XmN_Traversal_On
- XmN_unit_Type
- XmN_value
- XmN_value_Changed_Callback
- XmN_Vertical_Scroll_Bar
- XmN_Visible_Item_Count
- XmN_width
- XmN_X
- XmN_Y
- Xm_1000TH_Inches
- Xm_Add_Tab_Group
- Xm_Alignment_Beginning
- Xm_Alignment_Center
- Xm_Any_Callback_Struct_Ptr
- Xm_Attach_Form
- Xm_Attach_Widget
- Xm_Automatic
- Xm_Command_Get_Child
- Xm_Compound_String
- Xm_Constant
- Xm_Create_Bulletin_Board_Dialog
- Xm_Create_Cascade_Button
- Xm_Create_Command
- Xm_Create_Error_Dialog
- Xm_Create_Form
- Xm_Create_Frame
- Xm_Create_Information_Dialog
- Xm_Create_Label
- Xm_Create_Main_Window
- Xm_Create_Menu_Bar
- Xm_Create_Option_Menu
- Xm_Create_Pulldown_Menu
- Xm_Create_Push_Button
- Xm_Create_Row_Column
- Xm_Create_Scale
- Xm_Create_Scrolled_List

- Xm.Create.Scrolled.Window
- Xm.Create.Selection.Box
- Xm.Create.Separator
- Xm.Create.Text
- Xm.Create.Toggle.Button
- Xm.Create.Warning.Dialog
- Xm.Dialog.Cancel.Button
- Xm.Dialog.Command.Text
- Xm.Dialog.Help.Button
- Xm.Dialog.List
- Xm.Dialog.Selection.Label
- Xm.Double.Line
- Xm.Horizontal
- Xm.Label.Widget.Class
- Xm.List.AddItem.Unselected
- Xm.List.Callback.Struct
- Xm.List.Callback.Struct.Ptr
- Xm.List.Delete.Pos
- Xm.List.Deselect.All.Items
- Xm.List.Set.Pos
- Xm.Main.Window.Set.Areas
- Xm.Max.On.Top
- Xm.Message.Box.Get.Child
- Xm.One.Of.Many
- Xm.Pack.None
- Xm.Pack.Tight
- Xm.Pixmap
- Xm.Push.Button.Widget.Class
- Xm.Scroll.Bar.Get.Values
- Xm.Scroll.Bar.Set.Values
- Xm.Selection.Box.Get.Child
- Xm.Separator.Widget.Class
- Xm.Single.Line.Edit
- Xm.Single.Select

- Xm.Static
- Xm.string.context
- Xm.String.Create.L.To.R
- Xm.String.Default.Charset
- Xm.String.Free
- Xm.Text.Get.String
- Xm.Toggle.Button.Callback.Struct.Ptr
- Xm.Toggle.Button.Get.State
- Xm.Vertical

- X.Toolkit.Intrinsics.OSF Identifiers

- Addr.To.Widget
- Arg.List
- Arg.List.Rec
- Cardinal
- Dimension
- Null.Address
- Null.Arg.List
- Null.Option.List
- Null.Widget
- Null.Widget.List
- Pixel
- Widget
- Widget.Class
- Widget.List
- Widget.To.Addr
- XEvent
- XtN.Resource.String
- Xt.Add.Callback
- Xt.Clear.Arg.List
- Xt.Clear.Widget.List
- Xt.Create.Managed.Widget
- Xt.Display

- Xt_Get_Values
- Xt_Initialize
- Xt_Main_Loop
- Xt_Make_Arg_List
- Xt_Make_Widget_List
- Xt_Manage_Child
- Xt_Manage_Children
- Xt_Move_Widget
- Xt_Parent
- Xt_Quit_X_Request
- Xt_Realize_Widget
- Xt_Set_Arg
- Xt_Set_Keyboard_Focus
- Xt_Set_Sensitive
- Xt_Set_Values
- Xt_Set_Widget
- Xt_UnManage_Child
- Xt_Unmanage_Children
- Xt_Window

B.2 SAIC Binding Library Packages

- X_Windows Identifiers
 - Bits
 - Bit_Data_Array
 - Bytes
 - Byte_Array
 - Coordinate
 - Create_Pixmap
 - Depth_Type
 - Display
 - Drawable
 - Free_Pixmap
 - Graphic_Output.Bitmap_File_Invalid
 - Graphic_Output.Bitmap_No_Memory
 - Graphic_Output.Bitmap_Open_Failed
 - Graphic_Output.Bitmap_Status_Type
 - Graphic_Output.Copy_Plane
 - Graphic_Output.Create_Bitmap_From_Data
 - Graphic_Output.Create_GC
 - Graphic_Output.Create_Pixmap_From_Bitmap_Data
 - Graphic_Output.Draw_Rectangle
 - Graphic_Output.GC_Foreground
 - Graphic_Output.Gc_Mask_Type
 - Graphic_Output.Gc_Value_Record
 - Graphic_Output.Graphic_Context
 - Graphic_Output.Read_Bitmap_File
 - Lower_Window
 - Pixels
 - Pixmap
 - Plane_Mask
 - Point
 - Rectangle
 - Store_Name

Bibliography

1. Boehm, Barry W. "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, pages 61-72 (May 1988).
2. Booch, Grady, editor. *Software Components with Ada*. Menlo Park CA: Benjamin/Cummings Publishing Company, Inc., 1987.
3. Booch, Grady. *Software Engineering with Ada* (Second Edition). Menlo Park CA: Benjamin/Cummings Publishing Company, Inc., 1987.
4. Bullinger, H.J., editor. *Software Ergonomics: Advances and Applications*. West Sussex England: Ellis Horwood Limited, 1988.
5. Chikofsky, Elliot J. and James H. Cross II. "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, pages 13-17 (January 1990).
6. Cleveland, Scott. "Personal conversations," (Aug 1992).
7. Department of the Air Force. *US Air Force Basic Doctrine*. AFM 1-1. Maxwell AFB AL: Air University, March 1984.
8. Foley, James D. and others. *Computer Graphics, Principles and Practice* (Second Edition). Reading MA: Addison-Wesley Publishing Company, 1990.
9. Grudin, Jonathan. "The Case Against User Interface Consistency," *Communications of the ACM*, 32:1164-1173 (October 1989).
10. Horton, Capt Andre M. *Design and Implementation of a Graphical User Interface and a Database Management System for the Saber Wargame*. MS thesis, AFIT/GCS/ENG/91D-08, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
11. Johnson, Eric F. and Kevin Reichard. *Power Programming ... Motif*. Portland OR: MIS Press, 1991.
12. Klabunde, Gary W. *An Ada-based Graphical Postprocessor for the Saber Wargame*. MS thesis, AFIT/GCS/ENG/91D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
13. Kross, Capt Mark S. *Developing New User Interfaces for the Theater War Exercise*. MS thesis, AFIT/GCS/ENG/87-19, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987 (AD-A189744).
14. Lawlis, Patricia. Class handout distributed in CSCE 595, Software Generation and Maintenance. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, April through June 1992.
15. Mann, Capt William F. III. *Saber: A Theater Level Wargame*. MS thesis, AFIT/GOR/ENS/91M-9, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1991 (AD-A238825).
16. Molich, Rolf and Jakob Nielsen. "Improving a Human-Computer Dialogue," *Communications of the ACM*, 33:338-348 (March 1990).

17. Ness, Capt Marlin A. *A New Land Battle for the Theater War Exercise*. MS thesis, AFIT/GE/ENG/90J-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, June 1990 (AD-A223087).
18. Nielsen, Jakob. "Traditional Dialogue Design Applied to Modern User Interfaces," *Communications of the ACM*, 33:109-118 (October 1990).
19. Quick, Darrell. *A Graphics Interface for the Theater War Exercise*. MS thesis, AFIT/GCS/ENG/88D-16, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988 (AD-A205902).
20. Roth, Mark A. "Personal conversations," (January through November 1992).
21. Rumbaugh, James and others. *Object-Oriented Modeling and Design*. Englewood Cliffs NJ: Prentice Hall, 1991.
22. Sherry, Capt Christine. *Object-Oriented Analysis and Design of the Saber Wargame*. MS thesis, AFIT/GCS/ENG/91D-21, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
23. Software Productivity Consortium. *Ada Quality and Style: Guidelines for Professional Programmers*. Technical Report. Herndon VA, 1991.
24. Sommerville, Ian. *Software Engineering*. Reading MA: Addison-Wesley Publishing Company, 1982.
25. Systems Engineering Research Corporation. *SA-Motif Installation Guide*. Technical Report. Mountain View CA, October 1991.
26. Systems Engineering Research Corporation. *SA-Motif Release Notes*. Technical Report. Mountain View CA, October 1991.
27. Systems Engineering Research Corporation. *SA-Motif User's Manual*. Technical Report. Mountain View CA, October 1991.
28. Young, Douglas. *The X Window System: Programming and Applications with Xt (OSF/Motif Edition)*. Englewood Cliffs NJ: Prentice Hall, 1990.
29. Zave, Pamela. "The Operational versus the Conventional Approach to Software Development," *Communications of the ACM*, 27:101-115 (February 1984).

Vita

Captain Donald R. Moore was born on 29 December, 1953 in Tupelo, Mississippi. He graduated from Itawamba Agricultural High School in Fulton, Mississippi and attended Mississippi State University at Starkville, Mississippi. He was graduated magna cum laude in 1986 with a Bachelor of Science degree in Computer Science. Later in 1986, he was graduated as a Distinguished Graduate from Officer's Training School at Lackland AFB, Texas and attended the Communications and Computer course at Keesler AFB, Mississippi. He was then assigned to the 4441st Tactical Training Group, Hurlburt Field, Florida as a Tactical Simulation Branch Chief. He entered the School of Engineering at the Air Force Institute of Technology in May, 1991.

Permanent address: Route 1, Box 216A,
Greenwood Springs, MS
38848

December 1992

Master's Thesis

**AN ENHANCED USER INTERFACE
FOR THE SABER WARGAME**

Donald R. Moore, Capt, USAF

Air Force Institute of Technology
Wright-Patterson AFB, OH 45433-6583

AFIT/GCS/ENG/92D-10

AU CADRE/WG
Maxwell AFB, AL 36112

Approved for public release; distribution unlimited

This thesis is part of an on-going effort by the Air Force Institute of Technology to develop a computer-based, theater-level wargame for the Air Force Wargaming Center at Maxwell AFB, AL. The wargame, Saber, is intended to augment the education the students receive at the Air War College and the Air Command and Staff College in the employment of air and ground power. This thesis documents the integrated design and implementation of the two components of the Saber user interface: the pre-processor and the post-processor. Although previous thesis students designed and implemented substantial portions of the user interface, a fully operational interface was not completed for the end-user. In particular, the pre-processor design and implementation was incomplete and the user interface was constrained by its Ada to X Windows interface. Furthermore, the design and implementation of the two processors now needed to be integrated. The user interface was designed using object-oriented programming techniques. As necessary, reverse engineering techniques were used to extract the design of the existing implementation. Although the user interface application is implemented in the Ada programming language, it relies upon several software libraries including the X Window System and the OSF/Motif widget. Furthermore, software libraries from the Software Technology for Adaptable Reliable Systems (STARS) Foundation were used to provide the interface (the binding) between the Ada application software and the X Window System including Motif. This thesis also investigates the feasibility of replacing the STARS' bindings with those developed by the Systems Engineering Research Corporation.

Wargames, Ada, X Windows, Man Computer

112

UNCLASSIFIED

UNCLASSIFIED

UNCLASSIFIED

UL